ENCICLOPEDIA PRACTICA DE LA

# INFORMATICA APLICADA

31

## Los lenguajes de la Inteligencia Artificial

SINCLAIR AMSTRAD COMMODORE IBM

José Arteche




EDICIONES SIGLO CULTURAL

ENCICLOPEDIA PRACTICA DE LA

# INFORMATICA APLICADA

# 31

# Los lenguajes de la Inteligencia Artificial

EDICIONES SIGLO CULTURAL

*Una publicación de*

**EDICIONES SIGLO CULTURAL, S.A.**

Director-editor:
RICARDO ESPAÑOL CRESPO.

Gerente:
ANTONIO G. CUERPO.

Directora de producción:
MARIA LUISA SUAREZ PEREZ.

Directores de la colección:
MANUEL ALFONSECA, Doctor Ingeniero de Telecomunicación y Licenciado en Informática
JOSE ARTECHE, Ingeniero de Telecomunicación

Diseño y maquetación:
BRAVO-LOFISH.

Dibujos:
JOSE OCHOA Y ANTONIO PERERA.

Tomo XXXI. **Los lenguajes de la inteligencia artificial**
José Arteche, Ingeniero de Telecomunicación

Ediciones Siglo Cultural, S.A.

Dirección, redacción y administración:
Pedro Teixeira, 8, 2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid

Publicidad:
Gofar Publicidad, S.A. San Benito de Castro, 12 bis. 28028 Madrid.

Distribución en España:
COEDIS, S.A. Valencia, 245. Teléf. 215 70 97. 08007 Barcelona.
Delegación en Madrid: Serrano, 165. Teléf. 411 11 48.

Distribución en Ecuador: Muñoz Hnos.

Distribución en Perú: DISELPESA.

Distribución en Chile: Alfa Ltda.

Importador exclusivo Cono Sur:
CADE, S.R.L. Pasaje Sud América. 1532. Teléf.: 21 24 64.
Buenos Aires - 1.290. Argentina.



ISBN del tomo: 84-7688-114-2.
ISBN de la obra: 84-7688-018-9.

Fotocomposición:
ARTECOMP, S.A. Albarracín, 50. 28037 Madrid.

Imprime:
MATEU CROMO. Pinto (Madrid).



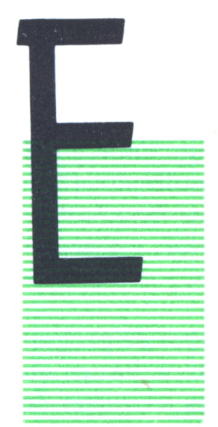
Depósito legal: M. 15.345-1987
Printed in Spain - Impreso en España.

Suscripciones y números atrasados:
Ediciones Siglo Cultural, S.A.
Pedro Teixeira, 8, 2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid
Mayo, 1987.
P.V.P. Canarias: 365,-

# INDICE

*A todos los que,*
*con su apoyo y paciencia*
*han hecho posible este libro:*
*las dos Blancas, Arancha,*
*Amaya, Paco y Ricardo...,*
*entre otros.*

# INTRODUCCION



STE libro es una introducción a los lenguajes que se suelen utilizar para programar las aplicaciones informáticas desarrolladas en las áreas que los expertos incluyen en el término genérico de Inteligencia Artificial (I.A.)

Lo primero que habría que preguntar es qué se entiende por I.A. Según Henry Bergson (*L'Evolution Créatrice,* 1907), «Inteligencia... es la facultad de construir objetos artificiales, especialmente herramientas, para fabricar herramientas». En el ámbito de la informática esto supondría no sólo crear sistemas (herramientas del hombre, al fin y al cabo) para procesar los datos o las informaciones, sino «sistemas que construyan sistemas».

Es decir, que, siguiendo esta línea de razonamiento, deberíamos entender que los «sistemas informáticos» aportan «inteligencia» cuando incluyen en alguna medida esa capacidad de evolución para generar diversos sistemas, adecuados a las distintas necesidades. Algunas de las aplicaciones que se consideran en el ámbito de la I.A. disponen de características de este estilo; especialmente los sistemas de monitorización y los sistemas expertos (sistemas de toma de decisiones variadas ante situaciones «nuevas» a partir de datos preexistentes).

Sin embargo, se acepta que son «inteligentes» otros procesos en que esta capacidad «creativa» no es tan evidente: en un sistema de procesamiento de palabra o en uno de visión artificial, priman las cualidades que hacen referencia a la posibilidad de evaluación de contextos complejos, al procesamiento rápido de gran cantidad de datos y a la flexibilidad de adaptación al entorno en el que se realizan los cómputos.

Quizá en estos casos fuera más adecuada la idea de Turing sobre la definición de «inteligencia» cuando proponía como prueba de un «sistema inteligente» la consistente en definir como tal a una máquina o sistema si su

«comportamiento» parecía semejante al humano a un observador exterior a los procesos que se realizan.

En cualquier caso, no es fácil abordar la problemática total de la I.A. de un solo «vistazo». En esta colección de libros han aparecido varios volúmenes (y algún otro está pendiente de publicación) sobre áreas específicas de la I.A. Todas ellas tienen en común, sin embargo, algunas técnicas básicas y los lenguajes informáticos en que se programan. El objeto del presente libro es estudiar, precisamente, estos lenguajes de programación de las aplicaciones de la I.A.

Se intenta obtener tres metas en este libro, bien que al nivel básico que corresponde al espacio disponible y al destino de la colección (sin que por ello deje de ser el trabajo efectuado absolutamente riguroso y desarrollado con todo nuestro interés): presentación general de los lenguajes utilizados en I.A. y sus características (donde subyacen, en cierto modo, las características generales que debe tener un sistema para ser incluido en el ámbito de la I.A.); descripción de dos de los lenguajes más utilizados (descripción básica pero suficiente, según entendemos, para poder llegar a programar en cualquiera de los dos); presentación de las técnicas elementales de programación que con estos lenguajes se desarrollan, y que subyacen en todas las aplicaciones de I.A.

En el primer capítulo se introduce la problemática general de programación para la I.A., en el segundo se explica la programación básica en LISP ("uno de los más antiguos y uno de los más modernos lenguajes de programación") y en el tercero la programación en PROLOG ("el lenguaje de programación del año 2000").

Como se incluyen numerosos ejemplos entendemos que sería útil seguir la lectura estando delante del ordenador (¡encendido!); pero esto no es imprescindible para entender los conceptos expuestos y obtener una visión clara de los «lenguajes de la I.A.».

# LOS LENGUAJES DE LA INTELIGENCIA ARTIFICIAL 1

## ¿SON NECESARIOS LENGUAJES ESPECIFICOS PARA LA INTELIGENCIA ARTIFICIAL?



LA primera pregunta que surge cuando se piensa en «los lenguajes de la I.A.» es: ¿realmente es necesario disponer de lenguajes específicos para programar tareas de las que solemos incluir en la I.A.?

La respuesta a esta pregunta es, según la mayoría de los expertos, que no es *necesario* disponer de ningún lenguaje específico para programar I.A., pero que *es muy conveniente*. En efecto, tal como hemos comentado, básicamente, la I.A. es un modo de enfocar los problemas, unas técnicas específicas de desarrollo de la programación, unos procesos concretos que con estas técnicas y con otros enfoques se pueden abordar, pero en informática, al fin y al cabo. Se puede decir que casi en el cien por cien de los casos es posible programar «cualquier cosa en cualquier lenguaje»; sin embargo, nadie duda que hay lenguajes especialmente aptos para ciertas tareas y otros para otras distintas.

De hecho, existen procesos y sistemas que deben incluirse en el ámbito de la I.A. y que están programados en lenguajes diversos. En concreto, al final de este capítulo, y como contraste con el resto de temas desarrollados en todo el libro, se presentan varias experiencias típicamente de I.A. (un «generador» de sistemas expertos, un sistema de interpretación en continuo de datos provenientes de sensores) escritos en otros lenguajes (Fortran, Pascal) distintos de los «típicos de Inteligencia Artificial» que se describen en estas páginas.

Como veremos, para representar adecuadamente el conocimiento y para operar con él (en los sistemas expertos), para representar y manejar eficientemente los fragmentos de lenguaje natural (en los sistemas de interface con el usuario o en los procesos de lenguaje natural), para almacenar y tratar con éxito la información gráfica, etc., es útil utilizar algún(os) lenguaje(s) con características especiales: incluso, dependiendo

del tipo de aplicación a abordar, será mejor utilizar uno u otro de los «lenguajes de la I.A.» disponibles.

Aportemos un argumento más a nuestro razonamiento subrayando cómo las empresas comerciales dedicadas a este área de la informática han tomado tradicionalmente el camino de la especialización (y no el de la generalización), llegando a construir máquinas específicas para los procesos de I.A. De esta idea han surgido las máquinas LISP y por este camino discurre, asimismo, el proyecto japonés de los «ordenadores de la quinta generación».

En efecto, la idea de construir máquinas específicas que se adecuaran a las estructuras que usualmente aparecen en los problemas de I.A. es muy antigua; ya en 1960 Newel y Tonge[1] escribieron: «La última versión disponible del lenguaje IPL (el IPL-VI) ha sido escrita como una propuesta de conjunto de instrucciones para un ordenador que ejecutaría directamente un lenguaje de proceso de la información y, por tanto, conseguiría una ejecución muchísimo más rápida que con la realización interpretativa disponible actualmente sobre máquinas convencionales.»

Las máquinas específicas para los procesos de I.A. se han desarrollado en dos direcciones distintas: como máquinas LISP y como ordenadores de proceso paralelo.

Las máquinas LISP son aquellas concebidas para implementar directamente las operaciones de proceso de listas y las funciones de manejo del almacenamiento de información proporcionadas por los intérpretes de LISP. Estas máquinas suelen disponer, además, de pantallas de presentación de alta calidad para facilitar el entorno de programación adecuado a estas tareas, con varias «ventanas» donde se examinan simultáneamente varios aspectos de un mismo proceso. En esta línea se han desarrollado numerosos proyectos y hay varias compañías dedicadas básicamente a esta actividad: Symbolics es una de las principales compañías suministradoras de productos de Inteligencia Artificial (69 millones de dólares en 1985, 1.250 máquinas LISP instaladas); y Lisp Machine Inc. (LMI). Estas dos compañías, junto con Rank Xerox (líder mundial por número de máquinas LISP instaladas), y Texas Instruments (que trabaja con licencia LMI), tienen monopolizado prácticamente el mercado mundial de máquinas LISP. Actualmente Sperry está introduciéndose con fuerza en este área y últimamente la misma IBM declara su interés por abordar este tipo de aplicaciones.

En la dirección de construir ordenadores de proceso paralelo (multiprocesadores) se mueve, por otro lado, el proyecto de «los ordenadores de la quinta generación» (además de que, en este proyecto, se haya tomado como base el lenguaje PROLOG y no el LISP).

---

[1] Newel, A., y Tonge, F. M.: «An Introduction to Information Processing Language V». *Communications of the ACM.* Vol. 3, págs. 205-211, 1960.

Sin embargo, la realidad del mercado informático actual con precios cada vez más bajos para máquinas con enormes potencias crecientes día a día e, incluso, la presión de algunas compañías (encabezadas por la propia IBM, según parece) hacen que haya que considerar como una opción muy válida la utilización de máquinas convencionales con eficaces intérpretes de lenguajes específicos de Inteligencia Artificial y sofisticados entornos de programación (con ventanas, menús, etc., y manejados desde ratones o pantallas táctiles).



## CARACTERÍSTICAS DE LOS LENGUAJES PARA LA INTELIGENCIA ARTIFICIAL

Aceptando, por tanto, que es muy útil utilizar algunos lenguajes específicos para resolver los problemas de I.A., surge la pregunta de cuáles son las características que deben tener los lenguajes que han de ser utilizados para la implementación de sistemas en el área de I.A.

La respuesta se puede dividir en dos partes (según E. Rich, pág. 390, obra citada):

— Las características que son importantes para la construcción de diversos tipos de sistemas informáticos complejos, incluyendo los de I.A.

- Las características que son particularmente importantes en la construcción de sistemas de I.A. debido a las propiedades especiales de dichos sistemas.

Consideremos primero las características importantes que debe poseer un buen lenguaje utilizado para la construcción de prácticamente cualquier gran sistema:

- Variedad de tipos de datos para describir los diferentes tipos de información necesarios en un sistema grande.

— Habilidad para descomponer el sistema en unidades pequeñas y claras, de manera que resulte relativamente fácil cambiar una parte del sistema sin tocar su totalidad.

— Estructuras de control flexible que faciliten la recursión y la descomposición paralela del sistema.

— Habilidad de comunicarse con el sistema interactivamente, para el desarrollo de programas y para obtener una efectividad máxima en el uso del sistema sobre el que se trabaja.

— Habilidad de producir código eficiente, de manera que el funcionamiento del sistema sea aceptable.

Resulta interesante constatar que, a pesar de que todas estas características son importantes hoy en día para cualquier gran sistema, fueron de-

tectadas como necesarias y desarrolladas en lenguajes de I.A. Por ejemplo, LISP fue, durante mucho tiempo, uno de los pocos lenguajes interactivos de gran aceptación, y se utilizó como una base para el desarrollo de técnicas de programación interactiva (Sandewall, 1978).

Además de estas características generales tan deseables, hay otras cuya importancia en el desarrollo de sistemas de I.A. es hoy en día reconocida, a pesar de que no han resultado útiles en otros contextos. Estas características incluyen:

— Herramientas especializadas para la manipulación de listas, ya que éstas son una estructura básica utilizada en todos los programas de I.A.

— Posibilidad de hacer ligaduras pospuestas *(late binding)* para informaciones como el tamaño de una estructura de datos o el tipo de objeto con el que se va a operar. En muchos programas de I.A. no es posible determinar estos elementos de otra manera que no sea «al vuelo», mientras el programa se está ejecutando. Por ejemplo, al construir una red semántica, normalmente no se sabrá con anticipación cuántos arcos emergerán de cada nodo.

— Facilidades para casar diseños *(pattern matching)* en la indentificación de datos y determinación del control. Estas facilidades para datos son una parte importante en el uso de grandes bases de conocimientos. La facilidad de «pattern matching» para control es básica en la ejecución y producción de sistemas.

— Facilidades para realizar cierto tipo de deducciones automáticas y para el almacenamiento y manejo de una base de datos de aserciones que forman la base para las deducciones.

— Herramientas para la construcción de estructuras de conocimiento complejas, como, por ejemplo, marcos *(frames)*, de manera que puedan ser agrupados y accedidos con una sola unidad ciertos bloques de información.

— Mecanismos mediante los cuales el programador puede introducir conocimientos adicionales que pueden ser utilizados por el sistema para enfocar su atención donde la rentabilidad parece que va a ser más alta.

— Estructuras de control que facilitan un proceso orientado hacia el objetivo en el proceso de arriba a abajo *(top-down)* junto con otro más convencional orientado hacia los datos (proceso de abajo a arriba) *(bottom-up)*.

— La habilidad de mezclar procedimientos y estructuras de datos declarativos del modo más conveniente a la tarea en particular que se quiere realizar.



# ALGUNOS LENGUAJES CONCRETOS

No existe ningún lenguaje que cumpla todas las características antes indicadas, sino que en alguno de ellos se pone especial énfasis en un aspecto concreto (en detrimento de otros) y en otros lenguajes se potencian otras cualidades.

Por otro lado, hay que señalar que la pléyade de lenguajes de programación existente en informática en general se convierte en una verdadera constelación cuando nos atenemos al área de la Inteligencia Artificial, como consecuencia lógica del enorme auge que esta materia ha experimentado en los últimos dos o tres lustros.

En efecto, entre los lenguajes básicos disponibles, sus diferentes dialectos y los entornos diversos de programación que engloban uno o varios de los anteriores (y los modifican y mejoran, en ocasiones) se pueden contar varias decenas de posibilidades para el usuario.

Uno de los más antiguos lenguajes de programación diseñados específicamente para aplicaciones de I.A. es IPL. De él derivó LISP, en la misma línea de lenguajes concebidos específicamente para el proceso de listas.

Posteriormente, a partir del año 1970, se desarrolló una segunda generación de lenguajes para la Inteligencia Artificial:

a) Por un lado, aparecieron numerosas versiones (e implementaciones específicas en máquinas) del LISP: FRANZLISP, INTERLISP, UCILISP, CMULISP, etc., y el COMMONLISP, que ha tratado de ser un estándar unificador de todos los dialectos existentes.

b) Por otro lado, se desarrolló el PROLOG (a partir de 1973, en la Universidad de Luminy, Marsella) y se han establecido posteriormente varias versiones e implementaciones de él.

c) Del LISP han derivado, asimismo, otros lenguajes como el PLANNER (del cual, a su vez, ha derivado CONNIVER), SMALLTALK, KRL, etc.

d) En otra dirección diferente, aunque derivado de LISP también, ha surgido un lenguaje de proceso de listas y gráficos que es el LOGO; utilizado básicamente en aplicaciones educativas.

Los últimos desarrollos en esta área a partir del año 1980 aproximadamente (algunos de los más recientes en fase de experimentación) se dirigen hacia la investigación de entornos únicos que engloben varios de los lenguajes antes indicados y hacia la creación de lenguajes de programación híbridos donde se combinan varias de las técnicas desarrolladas óptimamente en otros sistemas: entornos KEE, LOOPS, etc., y, más modernamente, COMMONLOG, VULCAN, etc.

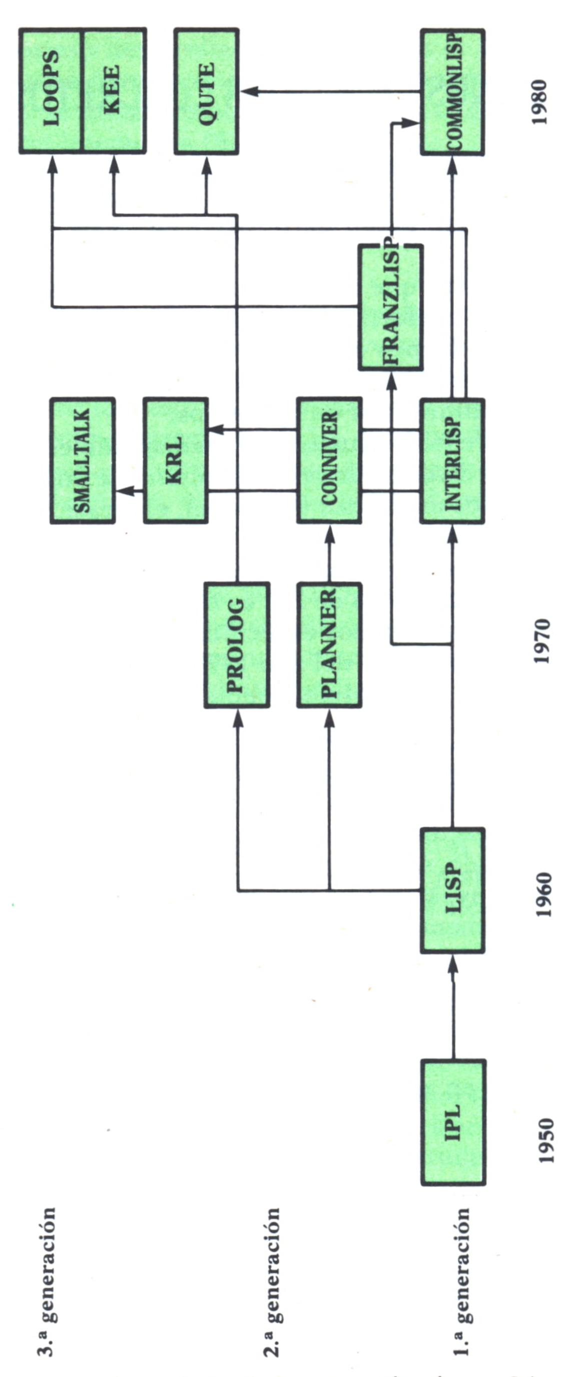



*Fig. 1. Genealogía de los lenguajes utilizados en I.A.*

Independientemente de la genealogía de los lenguajes de programación utilizados para la I.A., es útil considerar los diferentes grupos en que se pueden reunir, en función de los modelos de representación del conocimiento y de computación utilizados. En efecto, los lenguajes PROLOG y DUCK se utilizan básicamente para la programación lógica; SMALLTALK y FLAVORS en la programación orientada al objeto; OPS5 y EXPERT en la programación basada en sistemas de producción y, en fin, LOOPS y KEE combinan varios paradigmas y deben ser considerados sistemas híbridos.

| Programación lógica | Programación orientada al objeto | Sistemas de producción | Representación en marcos | Híbridos |
|---|---|---|---|---|
| PROLOG<br>DUCK<br>QUTE | SMALLTALK<br>FLAVORS | OPS5<br>EXPERT | KRL | LOOPS<br>KEE |

*Fig. 2. Los lenguajes usados en I.A., según el modelo de representación del conocimiento utilizado.*

En las páginas siguientes hacemos una breve presentación de algunos de los lenguajes más extendidos, dejando para los capítulos posteriores el análisis más profundo de los lenguajes básicos utilizados en las aplicaciones de I.A.: LISP (cap. 2) y PROLOG (cap. 3).

## IPL

Uno de los primeros lenguajes diseñados específicamente para aplicaciones de I.A. fue el IPL (Information Processing Language) [Newell, 1960]. Existe en la actualidad toda una serie de lenguajes IPL, de los cuales el IPL V ha sido el último en ser desarrollado. El principal objetivo de IPL fue el proceso de listas. El lenguaje en sí recordaba más a un lenguaje de máquina que a uno de alto nivel, como los que se usan actualmente. En la figura 3 se muestra un ejemplo de un programa IPL V. El programa considera un símbolo y una estructura de lista que representa un árbol, y comprueba si el símbolo existe o no en el árbol.

El IPL no está en uso actualmente. Ha sido completamente reemplazado por lenguajes de alto nivel como LISP, pero es interesante porque constituyó la base del desarrollo de los lenguajes de I.A.

| Nombre | PQ | Símb. | Unión | Comentario |
|---|---|---|---|---|
| EO | | J50 | | Desplaza WO y mueve el símb. a comprobar a WO. |
| 9-3 | | J60 | | Localiza la próxima celdilla (cell) del árbol. |
| | 70 | 9-1 | | Si no hay más celdilla sale con H5+. |
| | 12 | HO | | Da entrada al símb. en la siguiente celdilla de la lista. |
| | 11 | WO | | Da entrada al símb. a comprobar. |
| | | J2 | | Comprueba si los símb. son iguales. |
| | 70 | 9-2 | | Si son iguales sale con H5+. |
| 9-1 | 30 | HO | J30 | Desecha la referencia de la lista, saca WO. |
| 9-2 | 12 | HO | | Vuelve a dar entrada al símb. de la lista. |
| | | J132 | | Comprueba si es local. |
| | 70 | 9-3 | | Si no es local continúa a través de la lista. |
| | 12 | HO | | Da entrada al símb. de la lista otra vez. |
| | | J131 | | Comprueba si terminan los datos de nombre. |
| | 70 | | 9-3 | Si es el final de los datos continúa a través de la lista. |
| | 12 | HO | | Si no es el final de los datos, da nombre a una sublista. |
| | 11 | WO | | Da entrada al símb. a comprobar. |
| | | EO | | Aplica este proceso a la sublista. |
| | 70 | 9-3 | 9-1 | Si se encuentra en la sublista, sale con H5+. |

*Fig. 3.*

## PLANNER

PLANNER (Hewitt, 1971) es un lenguaje construido sobre LISP y diseñado para representar tanto la forma tradicional de razonar (hacia adelante) como la orientada al objetivo (hacia atrás). Los programas en PLANNER consisten en dos tipos de enunciados:

— Afirmaciones, las cuales simplemente exponen un hecho.

— Teoremas, los cuales describen cómo se puede inferir un hecho nuevo a partir de uno conocido.

A continuación hay ejemplos de los tipos de conocimiento que pueden ser representados por afirmaciones en PLANNER:

```
(AJO OLOROSO)
(PARTE BRAZO PERSONA)
(PARTE MANO BRAZO)
```

Hay tres clases de teoremas que pueden encontrarse en un programa PLANNER:

— Teoremas consecuentes que describen razonamiento orientado hacia atrás o hacia el objetivo. Un ejemplo es:

```
[CONSECUENTE
   (PARTE $?X $?Z))
   (OBJETIVO (PARTE $?X $?Y))
   (OBJETIVO (PARTE $?Y $?Z))]
```

Este teorema dice que si se quiere demostrar que X es parte de Z, se puede hacer encontrando una Y donde X es parte de Y e Y es parte de Z.

— Teoremas antecedentes, que describen razonamiento hacia adelante, hacia los datos.

Ejemplo de un teorema antecedente:

```
[ANTECEDENTE
   (PARTE $?X $?Y)
   (OBJETIVO (PARTE $?Y $?Z))
   (AFIRMACION (PARTE $?X $?Z))]
```

Este teorema dice que si se ha demostrado que X es parte de Y, entonces se buscan nuevas relaciones entre las partes que estén implicadas por ésta. Para hacerlo, se mira si hay algún valor Z donde Y es parte de Z. Si se encuentra alguno, concluir que X es parte de Z y añadir el hecho al conjunto de afirmaciones utilizables.

— Borrado de teoremas, elimina afirmaciones de la base de datos.

El lenguaje PLANNER nunca fue completado, pero sí un subconjunto de él, el micro-PLANNER. Varios programas de resolución de problemas se basaron en este subconjunto.

Una de las dificultades principales que aparecieron con PLANNER fue que el único modo de estructura de control disponible era retroceder el camino *(backtracking)* y este proceso era automático en vez de controlado por el programador. Esta restricción resultó ser un gran inconveniente y producía una disminución de eficiencia. Como remedio se construyó un nuevo lenguaje, el CONNIVER (Sussman, 1972). Un programador de CONNIVER puede explícitamente dirigir el control del flujo del programa.

## KRL

KRL (Bobrow, 1977) es un lenguaje construido sobre INTERLISP y que facilita la representación de conocimientos en marcos de discernimiento *(frame structures)*. El desarrollo de este lenguaje fue motivado por los siguientes supuestos sobre representación de conocimientos y por la necesidad de desarrollar los programas que los utilicen:

— Los conocimientos deben ser organizados alrededor de entidades conceptuales con descripciones y procedimientos asociados.

— Una descripción debe representar conocimientos parciales sobre una entidad y acomodar descriptores múltiples, los cuales pueden describir la entidad asociada desde diferentes puntos de vista.

— Un método importante de descripción es la comparación de una cantidad conocida, con especificaciones adicionales del ejemplo descrito, con respecto al prototipo.

— El razonamiento es dominado por un proceso de reconocimiento por el cual son comparados nuevos objetos y sucesos con conjuntos de prototipos esperados, y en los cuales se ligan (por medio de una clave) a estos prototipos ciertas estrategias de razonamiento especializado.

```
[Viaje UNIT Abstracto
   <SELF (un Suceso)>
   <modo (OR Avión Auto Bus)>
   <destino (una Ciudad)>]

[Visita UNIT Especialización
   <SELF (un ContactoSocial)>
   <visitante (una Persona)>
   <visitados (GrupoDe (una Persona))>]

[Suceso137 UNIT individuo
   <SELF {(una visita con
      visitante = Jaime
      visitador = (Artículo Juan María))

        (un viaje con
      destino = San Sebastián
      modo = Avión)}>]
```

*Fig. 4. Tres unidades (UNIT's) en KRL.*

— Los programas inteligentes requerirán procesos activos múltiples con horarios explícitos proporcionados por el usuario y heurísticas de asignación de recursos.

— La información debe ser agrupada para reflejar el uso de procesos cuyos resultados son efectuados por limitaciones de recursos y diferencias en la accesibilidad de la información.

— Un lenguaje de representación de conocimientos debe suministrar un conjunto flexible de herramientas, en vez de incorporar compromisos específicos sobre estrategias de proceso o la representación de áreas específicas de conocimientos.

Cada entidad en KRL es representada por una UNIT. Algunas UNITs representan conceptos abstractos; otras representan ejemplos específicos de esos conceptos. Una UNIT puede definir un objeto desde diferentes puntos de vista. En la figura 4 aparecen ejemplos de UNITs en KRL. Las UNITs del Viaje y la Visita representan conceptos abstractos, mientras que la UNIT de Suceso137 representa un ejemplo particular de un concepto. En realidad, es un ejemplo de dos conceptos, Viaje y Visita.

## LOOPS

El entorno LOOPS, desarrollado por Xerox, es un sistema de programación que incluye varios paradigmas de representación del conocimiento. LOOPS es una extensión de INTERLISP-D (la versión Xerox de INTERLISP). Aunque su nombre parece sugerir otra cosa (LOOPS = Logic Object Oriented Programming System = Sistema de Programación Orientado al Objeto), combina los siguientes estilos de programación: orientada a los procedimientos, orientada al objeto, orientada al dato y orientada a reglas. En el momento actual es aún (como confiesa la propia Rank Xerox) un prototipo de investigación, aunque lleva tres años siendo experimentado.

Sus características son las siguientes (respecto de los cuatro tipos de programación indicados):

*Programación orientada hacia el procedimiento*

En este paradigma se construyen grandes procedimientos basándose en otros más pequeños, mediante el uso de subrutinas. Los programas y datos se mantienen separados. La mayoría de los lenguajes de informática son de este estilo. La parte de LOOPS que está orientada hacia procedimientos es INTERLISP-D (Teitelman, 1978; Xerox, 1982). INTERLISP-D aparece en la base del logotipo de LOOPS para sugerir que proporciona la base sólida sobre la cual se construye LOOPS (ver fig. 5).

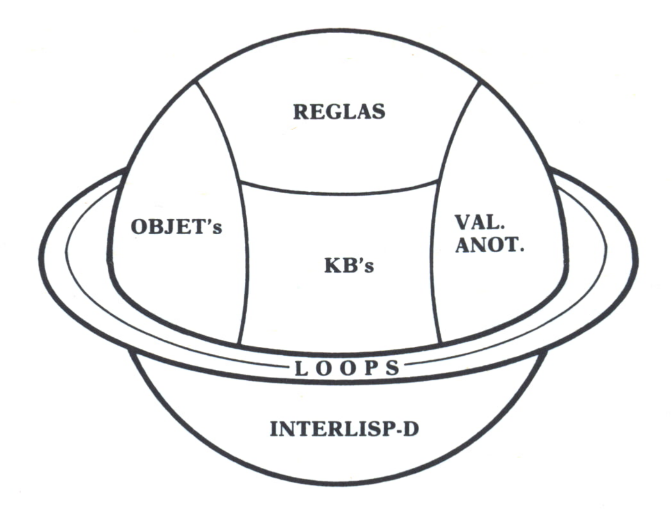



*Fig. 5. LOOPS según Stefik y Conway*.*

*Programación orientada hacia el objeto*

En este paradigma la información está organizada en «objetos», en los cuales se combinan instrucciones y datos. Los grandes objetos se construyen basándose en otros más pequeños. Los objetos se comunican entre sí mandándose «mensajes». Las normas para comunicarse con un objeto mediante el uso de mensajes constituye los protocolos de mensajes. Los protocolos estandarizados permiten a diferentes clases de objetos responder a la misma clase de mensajes. La «herencia» en una clase (dentro de una estructura en red) permite la especialización de objetos.

*Programación orientada al acceso*

Este paradigma es muy práctico para programas que controlan otros programas. Su mecanismo básico es una estructura llamada «valor activo», la cual tiene procedimientos que son llamados cuando se accede a ciertas variables. Una buena forma de imaginarse los valores activos es como sondas que pueden ser colocadas en las variables de objeto de un programa LOOPS. Estas sondas pueden activar procesos o cálculos adicionales cuando los datos son cambiados o leídos. Por ejemplo, pueden manejar indicadores que muestran los valores de las variables gráficamente.

*Programación orientada hacia reglas*

Este paradigma está especializado en la representación de los conocimientos para la toma de decisiones en un programa. En LOOPS, las reglas

* Del Centro de Investigación de Xerox de Palo Alto, California (EE.UU.).

son organizadas en «conjuntos de reglas» *(rules sets)* que especifican las reglas, una estructura de control y otras descripciones de las reglas.

## LOGO

El LOGO es un lenguaje surgido de los trabajos que Samuel Papert desarrolló en el MIT (Massachusetts Institute of Technology) con la finalidad de disponer de un lenguaje sencillo de aprender por los niños, pero que fuera poderoso en cuanto a sus capacidades para manejar información.

Se puede considerar un dialecto de LISP que, como éste, ha sido concebido para el manejo cómodo de listas. La programación en LOGO permite al usuario olvidar los detalles de la estructura del programa o de los datos y centrar su interés en la representación del conocimiento y su proceso. La sintaxis del LOGO es muy sencilla.

Las diferencias entre LOGO y LISP son básicamente dos: la mayor capacidad gráfica de LOGO y la simplicidad del lenguaje. En efecto, en LOGO se puede manejar un cursor (suele ser un triángulo o una flecha y se le llama «tortuga», cuando se presenta a los niños) que, mediante unos sencillos comandos, se hace mover por la pantalla. Los comandos son claros y directos y, como en LOGO se pueden redefinir fácilmente, es usual manejarlos en castellano: así se dice AV por «avanza», GD por «gira derecha», GI por «gira izquierda», etc. Junto con esta simplicidad de los comandos se dispone de la sintaxis cómoda de LOGO: la mayor ventaja es la eliminación de los paréntesis, tan engorrosos en LISP. Por todas estas razones, se puede aprender LOGO muy fácilmente, de un modo interactivo y sin ayuda de ningún profesor: esta es la razón de que se utilice tanto en la alfabetización informática de los niños.

Desde el punto de vista informático son características de LOGO la facilidad de definición y manejo de funciones y la comodidad de recursión: en efecto, cuando una función o rutina se llama a sí misma, los parámetros que el sistema guarda para devolver (al salir de la rutina o función llamada) se manejan de un modo mucho más ágil de lo que es usual en los otros lenguajes de programación (a veces se desechan ciertos valores, incluso), con lo que el proceso es simple y puede llegar a una recursión infinita.

Por ejemplo, para mover los discos en un proceso de paso de una pila (INICIAL) a otra (FINAL) en la resolución del problema de las «torres de Hanoi» se puede definir el siguiente programa LOGO:

```
PARA TORRES :NUMERO :INICIAL :MEDIA :FINAL
SI :NUMERO = 0 [DETENTE]
```

```
TORRES :NUMERO – 1 :INICIAL :FINAL :MEDIA
MOVER.DISCO :INICIAL :FINAL :MEDIA
TORRES :NUMERO – 1 :MEDIA :INICIAL :FINAL
FINAL
```

donde hemos utilizado los comandos castellanos PARA, SI y DETENTE (en vez de los correspondientes términos ingleses TO, IF y STOP) para definir la función TORRES (comando PARA = TO), para establecer el condicional (comando SI = IF) y para detener el programa cuando haya concluido el proceso de movimiento de los discos de una base a la otra. Como habrá observado el lector que conozca el lenguaje LOGO, se ha utilizado en el programa el siguiente proceso: se ha descompuesto la tarea de pasar el número correspondiente (:NUMERO) de discos de la torre INICIAL a la FINAL (usando la varilla MEDIA) en los siguientes tres pasos: desplazamiento de NUMERO - 1 discos desde la torre INICIAL a la MEDIA, paso del último disco desde la torre INICIAL a la FINAL y superposición de los NUMERO - 1 discos que estaban en la torre MEDIA sobre el que está en la torre FINAL. Naturalmente, para los desplazamientos de los NUMERO – 1 discos que se han indicado, hay que volver a llamar al propio procedimiento TORRES. El comando MOVER.DISCO que aparece en la definición de TORRES debe ser definido aparte y deberá realizar la tarea de eliminar un disco de la torre que se le da como primer parámetro (:INICIAL en el caso que hemos descrito) y ponerlo en la torre FINAL.

Como se ve, el lenguaje LOGO es fácil de comprender y programar, los comandos son sencillos y directos, la definición de las funciones se realiza de un modo muy simple y no hay dificultad ninguna para definir un proceso recurrente.

## KEE

El sistema KEE (Knowledge Engineering Environment, Entorno de Ingeniería del Conocimiento) es un entorno de programación desarrollado por Intelli Corp., una de las compañías más avanzadas en el desarrollo de software y «herramientas» en el área de I.A., en el mundo. Además, ha sido incorporado por Sperry como software básico en sus estaciones de trabajo para I.A. (sistemas Explorer), por Xerox en sus sistemas de la serie 1100 (diseñados específicamente para trabajar en aplicaciones del área de la I.A.), y por otras compañías.

Tiene características muy interesantes para preparación de pequeños sistemas expertos y prototipos, útiles en Ingeniería del Conocimiento, pero

también dispone de lenguajes de programación y herramientas útiles para el ingeniero experto.

Incorpora un sistema gráfico muy flexible que proporciona facilidades para la introducción de datos y para su actualización, así como para la presentación posterior (gráfica también) de la base de datos preparada, de los esquemas y reglas de producción introducidos, etc. Con la primera de estas facilidades enumeradas, se pueden construir cómodamente prototipos de sistemas complejos, con ayudas para ir visualizando los pasos ya dados y las informaciones aportadas. Pero, por otro lado, la presentación gráfica de los datos hace que sea muy claro el manejo posterior de los pequeños modelos ya creados. El sistema dispone de ayudas para la interpretación de la información, la partición de los problemas a resolver, el análisis de los posibles efectos laterales que se puedan producir y las consecuencias de las decisiones que se piense tomar, para la exploración de decisiones alternativas, etc.

Sus características, sin embargo, permiten que los ingenieros expertos puedan desarrollar sistemas más complejos con este software; en efecto, KEE proporciona varias metodologías muy experimentadas en I.A. que incluyen la programación orientada al objeto, la representación del conocimiento basada en marcos, la herencia taxonómica, el razonamiento basado en reglas, el razonamiento dirigido a los datos, un entorno LISP para la programación general, etc.

## OPS5

OPS5 es un lenguaje concebido específicamente para construir sistemas (especialmente sistemas expertos) basados en reglas (sistemas de producción). El nombre de OPS5 significa «Official Production System. Versión 5», y el epíteto de «Official» tiene una clara connotación irónica. El modelo de «sistema de producción» (para el que está diseñado el OPS5) está especialmente indicado cuando el conocimiento que se va a representar aparece de un modo natural en forma de reglas (de producción), cuando el control del programa es extremadamente complejo o cuando se espera que el programa vaya a ser modificado (de una manera importante a lo largo de su período de vigencia (ciclo de vida)). La diferencia básica entre los sistemas de reglas de producción y los modelos procedurales (clásicos de los lenguajes de programación convencionales) es la estructura de la información que se le ofrece en lo que se suele llamar programa: en los modelos procedurales un programa es una secuencia de unidades básicas (llamadas «instrucciones») que se ejecutan en un orden establecido (aunque se establezcan saltos o bifurcaciones), mientras que en el modelo de «sistema de reglas de producción» el «programa» consiste en una colección desordenada (exactamente, sin orden) de unidades básicas llama-

das «producciones», «reglas de producción» o, simplemente, «reglas». Por ejemplo, un conjunto de reglas podría tener el siguiente aspecto (escritas en castellano pero al estilo OPS5):

```
Regla 1: Servicio: inmediato
  IF (si) tipo de cliente es bueno
    AND (y) valor del pedido es alto
      AND (y) urgencia solicitada es urgente
  THEN (entonces) poner categoría de servicio a inmediato

Regla 2: Servicio: inmediato
  IF (si) tipo de cliente es bueno
    AND (y) valor de pedido es bajo
      AND (y) urgencia solicitada es urgente
  THEN (entonces) poner categoría de servicio a inmediato

Regla 3: Servicio: medio plazo
  IF (si) tipo de cliente es malo
    AND (y) urgencia solicitada es urgente
  THEN (entonces) poner categoría de servicio a medio plazo

Regla 4: Servicio: medio plazo
  IF (si) tipo de cliente es bueno
    AND (y) valor de pedido es bajo
      AND (y) urgencia solicitada es no urgente
  THEN (entonces) poner categoría de servicio a medio plazo

Regla 5: Servicio: sin prisa
  IF (si) tipo de cliente es malo
    AND (y) ... etc.
```

Siguiendo a Brownston y Farrell (ver referencia bibliográfica) podemos decir que las ventajas e inconvenientes que aparecen con la programación de OPS5 son las siguientes:

| | |
|---|---|
| Expresividad | Las reglas expresan fácilmente las actuaciones básicas de un proceso de símbolos. |
| Sencillez de control | Una consecuencia del formato limitado es que todos los datos son guardados en una forma similar y sólo se necesita una simple deducción mecánica. Con el sistema de producción de formato limitado, como, por ejemplo, los que usan encadenamiento hacia atrás, el control no tiene que ser especificado explícitamente, evitando así proceso oculto. |

**Capacidad de reacción** — Una consecuencia de la separación de conocimientos y control es que el sistema de producción puede enfrentarse con situaciones no previstas. Se pueden conseguir influencias recíprocas no previstas (pero útiles) como resultado de la utilización de la base de conocimientos cuando se necesita, en vez de hacer uso de estos conocimientos de acuerdo con secuencias predeterminadas.

**Modularidad** — Una consecuencia de la limitada interacción entre reglas y de la simplicidad del control es que las reglas tienden a ser modulares. Esto facilita la capacidad de reacción del sistema; por otro lado, esta capacidad facilita tanto la explicación como la modificación de reglas.

**Modificabilidad** — Como los conocimientos son almacenados en unidades casi independientes, se pueden añadir reglas sin causar muchos efectos secundarios. Esto facilita el crecimiento del sistema (y también de alguna manera su degradación estética, cuando no se encuentran muchas reglas específicamente relevantes).

**Facilidad de producir especificaciones** — Otra consecuencia de la separación entre el conocimiento y el control y del formato restringido es que las reglas son relativamente fáciles de explicar, ya que cada una se refiere a un conjunto muy concreto de conocimientos. La facilidad de explicación también depende de la naturaleza de las acciones a explicar.

**Lectura de máquina** — Los formatos restringidos producen reglas que pueden ser leídas por la máquina. Esto hace posible el limitar la cantidad de modificaciones y explicaciones automatizadas, el control de consistencia y el aprendizaje.

**Aprendizaje** — Mediante un formato estándar las reglas pueden ser tratadas como datos y los programas pueden entender y manipular sus propias representaciones, haciendo posible tanto el aprendizaje como el conocimiento propio. Sin embargo, se produ-

| | |
|---|---|
| | cen influencias nocivas entre la estandarización de formatos y la flexibilidad con la cual pueden ser representados. |
| Paralelismo | Pueden a menudo ejecutarse independientemente unos módulos del sistema de otros, haciendo posible de esta manera la eficiencia de las computaciones paralelas. |

Se debe tener en cuenta que hay pocos modelos de sistemas de producción que soporten en el proceso en tiempo de ejecución *(runtime)* las capacidades necesarias para el fácil acceso a todas las ventajas citadas.

Se pueden, por otro lado, indicar algunas desventajas que hay que tener en cuenta al utilizar un sistema de producción como modelo de proceso de conocimiento:

| | |
|---|---|
| Dificultades de expresión | Si un sistema tiene un formato muy restrictivo los conocimientos no pueden siempre ser representados de una manera tan estructurada como un modelo de sistema de producción exige. También, las limitaciones en los operadores booleanos (lógicos) permitidos en las condiciones de las reglas pueden forzar a la creación de reglas múltiples que incluyen condiciones similares. |
| Control poco clarificado | Una consecuencia de la comunicación limitada entre reglas, la separación de control y conocimientos, y el tener una simple deducción mecánica, es que algunos diseños de control como secuencias y bucles complejos, son difíciles de especificar. Es más difícil expresar y conectar conceptos en grandes estructuras que en reglas simples, incluso cuando estos conceptos son esencialmente fáciles. |
| Comunicación no deseada entre reglas | A pesar de la independencia de las reglas (opcional), pueden surgir problemas por la aparición de comunicaciones inesperadas entre reglas. Estas intercomunicaciones son una consecuencia de la limitación de la comunicación y de la separación del control y los conocimientos. |

Comportamiento no transparente

Las intercomunicaciones limitadas y la naturaleza básica de las acciones de las reglas, además de la dificultad de localizar el control, hacen materialmente imposible el entender y predecir el comportamiento de los sistemas de producción que usan estrategias de resolución de conflictos o que introducen un complejo control dentro de las reglas. Esto también dificulta las explicaciones y modificaciones.

Dificultad de desarrollo

En parte, como resultado del comportamiento no transparente del sistema, y en parte porque están muy poco desarrollados los entornos de programación de algunos sistemas de producción, las facilidades para el desarrollo *(debugging)* en los lenguajes de sistemas de producción son mínimas.

Lentitud

La mayoría de los sistemas de producción se ejecutan a una velocidad uno a dos órdenes de magnitud más lenta que los programas orientados al procedimiento, porque el evaluador *(matcher)* tiene que reconsiderar toda la situación para encontrar reglas aplicables a cada ciclo.

## APLICACIONES DESARROLLADAS EN LENGUAJES CLASICOS

Como contraste con las aplicaciones descritas y los lenguajes específicos utilizados para la resolución de problemas de I.A., se comentan a continuación dos ejemplos de sistemas desarrollados con técnicas, conceptos y mentalidad de I.A. pero en lenguajes convencionales (procedurales), con una eficacia grande, como ya se ha comentado.

### EX-TRAN 7

Es un paquete diseñado para construir sistemas expertos desarrollado en lenguaje Fortran. EX-TRAN 7 significa EXPERT TRANSLATOR a Fortran 77. En efecto, se han codificado en este lenguaje el motor de inferencia de un sistema experto y el generador inductivo. El experto puede faci-

litar las reglas al sistema bien de un modo explícito o bien de un modo indirecto por medio de archivos o ejemplos. Las reglas se generan en Fortran 77 y pueden, por tanto, ser linkadas con rutinas adicionales externas en Fortran. La estructuración flexible de las reglas permite establecer entre dichas reglas una dependencia jerárquica en el sistema experto resultante. La eficacia del sistema es enorme y, de hecho, ya se han construido varios sistemas expertos (y se están explotando) utilizando EX-TRAN 7. Ha sido desarrollado por la compañía británica Intelligent Terminals Ltd.

Las características básicas de EX-TRAN 7 son las siguientes, según sus autores:

— Las soluciones generales (en base a reglas) por el problema a resolver, son simples y claras.

— La estructuración flexible de las reglas permite establecer una dependencia jerárquica muy útil. La estructura de las reglas es controlada por el usuario desde el archivo base de texto. Se puede modificar completamente la estructura del sistema experto cambiando el texto del archivo base, aunque se mantenga el mismo conjunto de reglas.

— Se pueden incluir en el conjunto subrutinas externas (en Fortran) para poder preparar más fácilmente la captura de datos, soportar el componente de control de atributos (parte «if» de las reglas «if-then»), supervisar la validez de la porción «then» de las reglas, etc. Incluso estas rutinas externas facilitadas (e «incluidas dentro de») al sistema experto puede realizar cálculos intermedios necesarios en la ejecución de las reglas.

— Las reglas necesarias para la solución del problema se le pueden proporcionar directamente a EX-TRAN 7 o bien puede el sistema inferirlas inductivamente a partir de un conjunto de decisiones ejemplo que se le facilite; para esta tarea está dotado el sistema de un módulo específico de «aprendizaje inductivo». Esta cualidad es esencial en algunos problemas para los que no existen (o los expertos no son capaces de formular explícitamente) reglas suficientes sobre «cómo hacerlo».

— EX-TRAN 7 genera automáticamente a partir de las reglas, código fuente en Fortran 77, de tal modo que (una vez compilado) puede ser utilizado en la versión final («runtime») del sistema experto.

— Existen comandos de usuario para dirigir la operación general del sistema (en todas sus fases) de un modo cómodo.

— En el sistema experto resultante se incorpora un conjunto de elementos de ayuda (bajo petición) a utilizar en el momento de su uso. Con ello se puede hacer más cómoda y clara la utilización del sistema experto y se pueden incluir, además, capacidades de autoaprendizaje para los menos expertos.

## MXA

Este sistema está concebido como un shell o sistema concha; es decir, es un sistema de interface de usuario. Fue creado como una herramienta para la investigación acerca de cómo aplicar las técnicas de los sistemas expertos a la fusión de datos de sensores para obtener una interpretación continua de las informaciones obtenidas de un conjunto diverso de sensores de tipo militar. MXA incorpora el modelo básico de funcionamiento de un grupo de «fuentes de conocimiento» que interactúan con una base de datos ubicada en la memoria principal del equipo que soporta a este sistema. Para esta aplicación de MXA cada fuente de conocimiento consiste en un grupo de reglas de condición-acción con el conocimiento procedural soportado por la habilidad de usar funciones y procedimientos en las reglas.

El lenguaje de representación del conocimiento es una extensión de Pascal y la gran ventaja de MXA es, precisamente, que el compilador correspondiente genera lenguaje fuente Pascal para que sea compilado también.

Los principales objetos que maneja el sistema son hipótesis. Cada una de ellas es una ocurrencia de un tipo específico de datos y tiene la forma de un registro Pascal.

# LISP 2

## INTRODUCCION



TAL como hemos comentado, LISP es el lenguaje por excelencia de la Inteligencia Artificial. Para el desarrollo de aplicaciones en este área de la informática se utilizan también otros lenguajes, pero de aplicación más restringida: PROLOG (al que dedicamos el próximo capítulo) aunque está concebido específicamente para desarrollar «programación lógica», ha adquirido un extraordinario auge por su capacidad de deducción, sobre todo en la programación de sistemas expertos y en el procesamiento del lenguaje natural (áreas de enorme popularidad, hoy en día); SMALLTALK se utiliza para la programación «orientada al objeto», cuando esta forma de representación del conocimiento es la más adecuada al problema que se va a abordar; LOOPS es un entorno de programación que incorpora LISP y PROLOG y otras muchas herramientas para una programación eficaz; e incluso, se han desarrollado sistemas específicos para la resolución de problemas concretos (programación de «sistemas expertos», específicamente). Sin embargo, LISP sigue siendo el lenguaje más universalmente extendido (más aún en EE.UU. que en Europa o Japón) para los desarrollos de Inteligencia Artificial en general.

LISP es un lenguaje simbólico (pues está pensado para procesar palabras y frases, listas), en contraposición a los lenguajes numéricos (aunque también maneja números, claro está).

Por otro lado, LISP es un lenguaje típicamente funcional: las actividades básicas son realizadas por funciones (porción de código que devuelve un objeto a partir de otro u otros objetos), bien sean funciones primitivas (definidas en el propio intérprete), bien funciones definidas por el usuario. Programar en LISP es, en gran medida, definir funciones adicionales y combinarlas con las ya existentes. Una de las características del lenguaje que suele agradar más al programador en LISP es, precisamente, la faci-

lidad y flexibilidad que proporciona el lenguaje para que el usuario defina funciones adicionales.

Desde luego, no es LIPS el único lenguaje simbólico en que se pueden definir fácilmente funciones adicionales: en LOGO, por ejemplo, se pueden realizar estas actividades también. Pero LISP dispone de primitivas muy potentes en el manejo de listas y estructuras de control de proceso bastante útiles. (Aunque LOGO, quizá, sea más cómodo, más simple y con mejores capacidades gráficas (?)).

En este capítulo vamos a presentar un LISP un poco genérico (parecido quizá a MacLISP y, en algunas cosas, a CommonLISP), pero que esperamos sea suficiente para una buena introducción a la programación en este interesante lenguaje. La versión de LISP de que disponga el lector no coincidirá totalmente con todos los detalles de la sintaxis que vamos a utilizar, pero podrá fácilmente detectar las pequeñas diferencias existentes (y a las que iremos aludiendo) consultando el manual que le haya facilitado el suministrador del intérprete que vaya a manejar.

Sería muy útil disponer de un intérprete de LISP para ir practicando los ejemplos que se dan en el texto, pero no es imprescindible para entender el contenido de esta presentación y tener una buena idea de cómo se puede programar en LISP y cuáles son las características básicas (y diferenciales) de este lenguaje.



## CONCEPTOS DE PROGRAMACION

Las informaciones que se manejan en lenguaje LISP, los datos, pueden ser elementos simples o listas; a los elementos simples se les llama también átomos. En ocasiones, cualquier dato que se procese en LISP (átomos y listas) es llamado «objeto», especialmente cuando se manejan conceptos de «programación orientada al objeto».

Un átomo es un conjunto de caracteres (caracteres alfabéticos, numéricos o signos especiales) sin separación alguna (sin incluir espacios).

En LISP son distintos, a todos los efectos, las mayúsculas de las minúsculas; y, por tanto, son distintos los identificadores NUEVOVALOR, nuevovalor, nuevo-valor, Nuevo-valor, NuevoValor, etc.

Los átomos pueden ser números o símbolos (también llamados identificadores). Algunas versiones sencillas de LISP sólo utilizan números enteros (con o sin signo).

Por el contrario, en los identificadores o símbolos se admite la presencia de caracteres numéricos o signos especiales. (A veces se exige que el primer carácter de un identificador sea alfabético). Los identificadores pueden ser constantes o variables (los números son constantes). Los iden-

tificadores constantes pueden ser, a su vez, constantes literales, nombres de función o predicados y símbolos especiales (como «T», que representa el valor true = verdad).

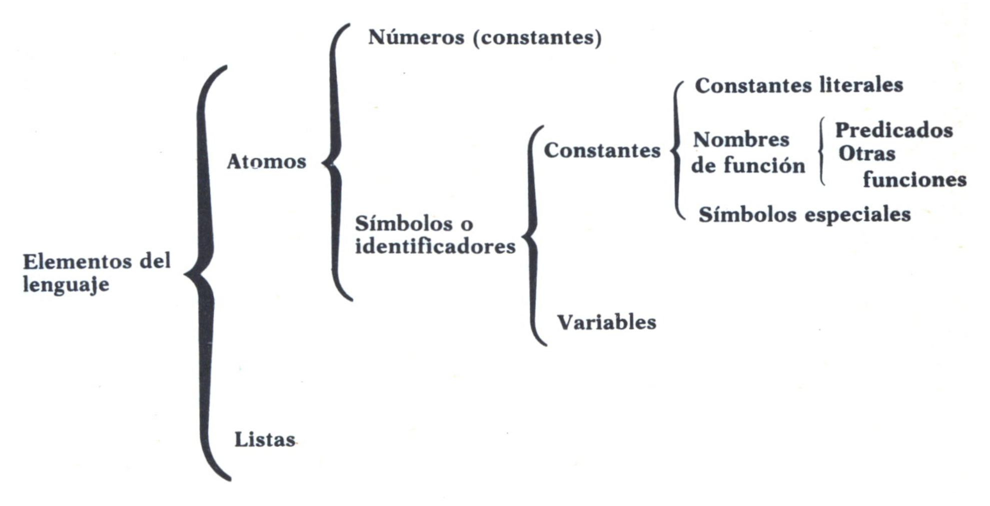



*Fig. 1. Elementos del lenguaje LISP.*

Una lista es un conjunto de elementos de lenguaje (átomos o listas) separados por espacios e incluidos en unos paréntesis. Una lista puede contener, a su vez, como elementos a otras listas. Existe la lista vacía, sin elementos (se escribe «()»).

Al conjunto de átomos y listas se le denomina S-expresión (o expresión-S, castellanizando el término) y significa «expresión simbólica»: el tratamiento de estas S-expresiones es el objeto del lenguaje LISP.



## Manejo de listas

Tal como hemos comentado, el elemento básico de datos en LISP es la lista. Una lista (sucesión de elementos separados por espacios y encuadrados entre paréntesis) puede contener, a su vez, listas; pero cada lista que incluye debe ser considerada como un único elemento. Así, la lista:

```
(ARTI X1 (A B C (X Y)) () M)
```

consta de cinco elementos: el tercero de ellos es una lista (la lista «(A B C (X Y))»), cuyo cuarto elemento es una lista de dos átomos; el cuarto elemento de la lista primera es la lista vacía (no tiene ningún elemento).

A veces, la lista vacía se representa por la constante «NIL», pues en la sintaxis del lenguaje son equivalentes.

Es importante, cuando se van a procesar listas, distinguir el primer elemento de la lista de los restantes elementos: el primer elemento se llama «cabeza» de la lista y el resto de elementos (ninguno, uno o varios) es la «cola» de la lista. Así, en la lista anterior la cabeza es «ARTI», mientras la cola es (X1 (A B C (X Y)) ((M).

Existen dos funciones primitivas de LISP para realizar la separación de estos dos elementos de una lista: son las primitivas «CAR» y «CDR». (Una función se llama «primitiva» si está definida de origen con el intérprete o compilador del lenguaje: las tareas básicas a realizar en LISP disponen de sus correspondientes funciones primitivas, aunque el usuario puede fácilmente —ya lo veremos— definir sus propias funciones. ¡Esa es una de las características que más agradan al programador de LISP!)

La función CAR devuelve el elemento (átomo o lista) que forma la cabeza de la lista. Así,

```
(CAR '(ARTI X1 (A B C (X Y)) () M))
```

devolverá el átomo «ARTI»; mientras que CDR (se lee «coulder», en inglés) devuelve una lista formada por los restantes elementos de la lista original:

```
(CDR '(ARTI X1 (A B C (X Y)) () M))
```

devolverá la lista:

```
(X1 (A B C (X Y)) () M)
```

(Obsérvese que hemos puesto un apóstrofo «'» delante de la lista que damos a «CAR» o «CDR»; este apóstrofo es imprescindible (tecléelo sin apóstrofo y comprobará que el intérprete le da un mensaje de error), pero su

significado se verá más adelante. Por ahora tómelo simplemente como un modo peculiar de escribir los argumentos de las funciones primitivas «CAR» y «CDR». Por otro lado, habrá observado que el conjunto total formado por la función —CAR o CDR— y su argumento, lo escribimos siempre entre paréntesis: en LISP ha de ser así).

Es útil decir expresamente que (como era de esperar, por otro lado):

```
(CAR '(UNICO))
```

devolverá el valor «UNICO», mientras que

```
(CDR '(UNICO))
```

devolverá el valor «NIL» (representando a la lista vacía; en latín «nada» se dice «Nihil» o «Nil»).

Por otro lado, si introducimos:

```
(CAR 'UNICO) o bien (CDR 'UNICO)
```

el intérprete dará un mensaje de error, pues ambas funciones necesitan un argumento en forma de lista (no un átomo).

También suele suceder que (CAR NIL) o (CDR NIL) den como resultado «NIL», pero hay versiones de LISP que dan error en esos casos. Subrayamos que las anteriores expresiones son equivalentes a (CAR ()) y (CDR ()), respectivamente. Además, obsérvese cómo NIL es a la vez un átomo (un símbolo o identificador especial) y una lista (no da error (CAR NIL), normalmente).

Por último, hay que decir que se pueden hacer llamadas consecutivas o «anidadas» a estas funciones. El intérprete admite la expresión:

```
(CAR (CDR '(A (B C) D)))
```

y da como resultado la lista (B C) (asegúrese de que sabe por qué); si, por el contrario, introducimos:

```
(CDR (CAR '((A (B C)) D)))
```

sucederá ahora que la primera evaluación (de «CAR») dará por resultado «(A (B C))» y el «CDR» posterior selecciona la cola (es decir, «(B C)») y la da *como una lista* ((B C)). (La diferencia es el doble paréntesis que aparece como resultado de «CDR»).

El anidamiento se puede llevar tan lejos como queramos y podemos, por tanto, escribir:

```
(CAR (CDR (CDR (CDR '(A B C D E)))))
```

para obtener el cuarto elemento de la lista, es decir «D».

Normalmente, el intérprete de LISP dispone de funciones especiales que sustituyen con seguridad y comodidad a los anidamientos de CAR y CDR; suele existir la función «CADR», definida como «el CAR del CDR de la lista»:

```
(CAR (CDR '(A B C)))
```

(es decir, B)

y la función «CDAR» que será:

```
(CDR (CAR '(A B C)))
```

Hay que observar que la expresión anterior dará error, pues el CAR de la lista es el átomo «A», y se da a CDR un átomo —no una lista— lo que produce error. Por el contrario:

```
(CDR (CAR '((A B) C)))
```

dará como resultado (B).

Del mismo modo existirán (o se definen fácilmente) las funciones CADAR, CADADR... hasta un límite de «A» y «D», que depende del intérprete que se esté usando.

Hay otra función muy útil en el manejo de listas que realiza la tarea opuesta a la «separación» de elementos de la lista: es «CONS», que permite «CONStruir» una nueva lista añadiendo un elemento delante de una lista dada. Los argumentos de la función primitiva «CONS» son un átomo y una lista y el valor resultante que se devuelve es una lista que tiene como cabeza el átomo y como cola el segundo elemento (lista) que se le ha dado. Así,

```
(CONS 'A '(B C))
```

dará como resultado (A B C), mientras que:

```
(CONS '(A B) '(C D))
```

devolverá ((A B) C D); es decir, siempre el primer argumento se toma como un átomo y el segundo como una lista a la que se añade, en cabeza, el átomo. Por tanto, se puede hacer:

```
(CONS 'A NIL)
```

y dará (A) o bien:

```
(CONS '(A) NIL)
```

que dará ((A)). También se puede hacer:

```
(CONS 'Ahora '(va bien))
```

para obtener (Ahora va bien), pero no daría el mismo resultado la expresión

```
(CONS '(Esto está) '(mucho peor))
```

pues devolverá ((Esto está) mucho peor).

Para comprender claramente por qué CDR devuelve la cola de la lista como una lista (aunque esté formada por un solo elemento), mientras que CAR devuelve la cabeza como un átomo, por qué CONS une un átomo (y no una lista) con una lista... y, en general, cómo se pueden procesar en LISP las listas, es útil ver de qué modo LISP considera y maneja las listas. La representación interna de las listas se hace en lo que se llama «cadena de celdas CONS», que son unos «casilleros» o «celdillas» donde se almacenan los diferentes elementos de la lista y los «punteros» o «indicadores» que señalan al siguiente elemento de la lista.

Sin embargo, es más clara (aunque no sea tan precisa) una representación en forma de «árbol binario», y será suficiente para nuestros propósitos. Un «árbol» es una representación de una estructura general de datos o de una secuencia de procesos o elementos, de tal modo que en cada nudo del árbol aparece uno de los elementos de unión y en cada rama las dos partes en que ese elemento de unión divide la secuencia dada. Por ejemplo, en la figura 2 aparece un árbol que puede representar la frase «Si tomamos un taxi y el taxi no pincha, llegaremos a tiempo». El árbol es «bi-

Sí

y

Tomamos

No pincha

Llegaremos a tiempo

un taxi

el taxi

*Fig. 2. Un árbol binario.*

nario» porque de cada nodo «no terminal» parten a lo sumo dos ramas. Un caso muy concreto de árbol binario es el que se puede utilizar para representar una lista.

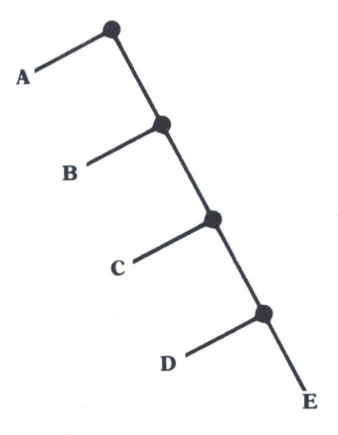



*Fig. 3. Arbol representativo de una lista plana.*

En la figura 3 aparece el árbol representativo de la lista (A B C D E), mientras que en la figura 4 aparece un árbol más complejo para la lista (más complicada) (A (B C) D (E F) G).

Sobre estos árboles establecemos la norma siguiente: «las ramas que están hacia la derecha son subárboles —y representan listas, por tanto— las ramas que están hacia la izquierda se consideran átomos en la composición o análisis

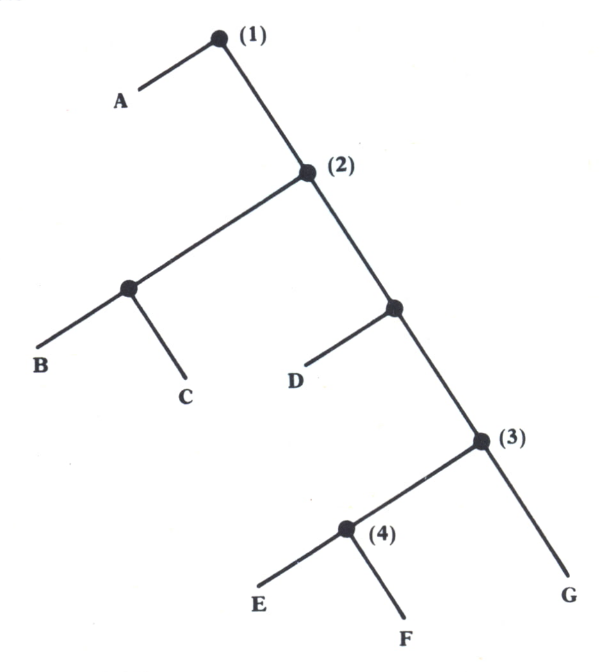


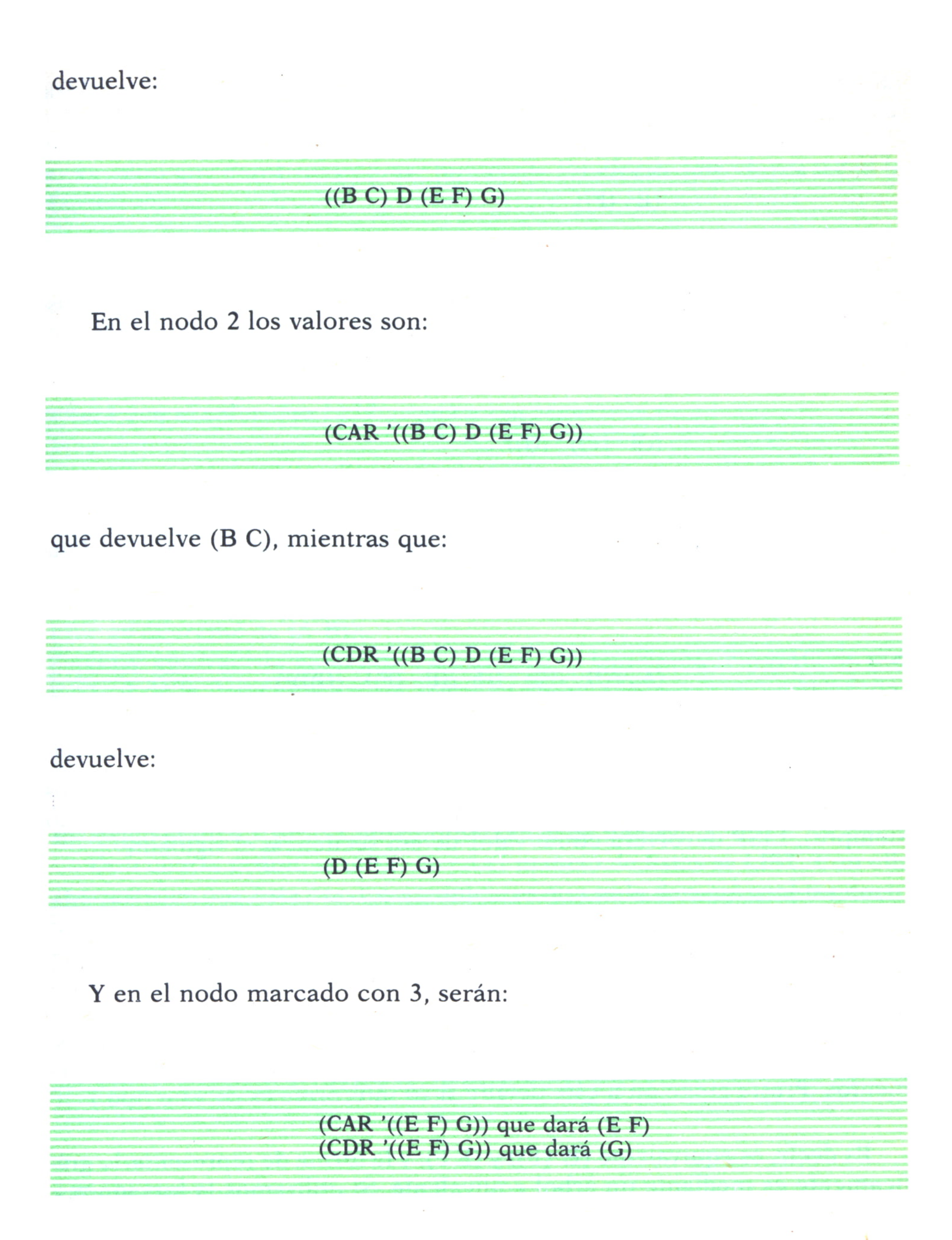

*Fig. 4. Arbol binario representativo de una lista jerárquica.*

—aunque sean, a su vez, listas—». La función CAR separa la rama de la izquierda en cualquier nudo y la función CDR separa la rama (o árbol) de la derecha. Así en el nodo 1 se tiene que:

```
(CAR '(A (B C) D (E F) G))
```

devuelve A, mientras que:

```
(CDR '(A (B C) D (E F) G))
```



devuelve:

```
((B C) D (E F) G)
```

En el nodo 2 los valores son:

```
(CAR '((B C) D (E F) G))
```

que devuelve (B C), mientras que:

```
(CDR '((B C) D (E F) G))
```

devuelve:

```
(D (E F) G)
```

Y en el nodo marcado con 3, serán:

```
(CAR '((E F) G)) que dará (E F)
(CDR '((E F) G)) que dará (G)
```

Mediante estos árboles se ve más fácilmente cómo se podría escribir de otro modo estas listas: en efecto, si consideramos el punto que aparece en cada nodo del árbol de la figura 4 como un conector binario (es

decir, que une dos operandos), el árbol que nace en el nodo 4 se podría escribir:

```
(E . (F))
```

(recuérdese que la rama de la derecha debe ser considerada árbol —lista— y debe ser puesta, por tanto, entre paréntesis y la de la izquierda no).

Del mismo modo, el árbol que nace a partir del nodo 3 será:

```
((E . (F)) . (G))
```

y por fin, el conjunto del árbol (representativo de la lista que considerábamos al principio):

```
(A . ((B . (C)) . (D . ((E . (F)) . (G)))))
```

En efecto, LISP admite esta otra representación; es más, si tecleamos en el ordenador la expresión «punteada» anterior (con un apóstrofo, como siempre), LISP devolverá como valor de esa secuencia el siguiente:

```
(A (B C) D (E F) G)
```

(Es muy común que el intérprete de LISP necesite que se ponga un espacio delante del punto y otro detrás, para separarlo del resto de los elementos. Preste atención a este detalle si su intérprete es de este tipo.)

Es decir, es equivalente (E. (F)) y (E F); y la forma de representarlo gráficamente es mediante un árbol equivalente. Si utilizamos la función primitiva CONS para «construir» este árbol o lista debemos escribir:

```
(CONS 'E '(F))
```

Sin embargo, se puede utilizar CONS de otro modo para dar lugar a la estructura que se llama «par punteado». Si escribimos el segundo parámetro de CONS no como una lista sino como un átomo, la función no dará error, sino que construirá un par punteado. En efecto, introduzca usted la expresión:

```
(CONS 'A 'B)
```

y verá que el valor que LISP asigna a esa secuencia es:

```
(A.B)
```

(que es diferente de (A B)).

(Esto equivaldría en el árbol a suprimir la condición impuesta de que la rama de la derecha debe ser considerada una lista —otro subárbol—.)

Esta nueva estructura de datos es muy útil en algunas ocasiones, en casos especiales. Aunque no entremos aquí en la utilización del par punteado frente a la lista y en las diferencias de utilización de CONS y LIST (primitiva que veremos a continuación), diremos que es muy útil en el análisis léxico-gráfico y procesamiento del lenguaje natural.

Otra cuarta primitiva utilizada en el manejo de listas es LIST. Con ella se construye una lista con los argumentos que se le dan, tal cual (es decir, si aparecen como listas, los incluye como listas; si aparecen como átomos, como átomos). La función LIST admite cualquier número de argumentos (no necesariamente dos, como CONS). De este modo, podemos hacer:

```
(LIST 'A '(B C) 'D '(E F) 'G)
```

y obtendremos como resultado:

```
(A (B C) D (E F) G)
```

Si escribimos:

```
(LIST 'Ahora '(va bien))
```

obtendremos:

```
(Ahora (va bien))
```

y, para que no aparezcan los paréntesis, deberíamos escribir:

```
(LIST 'Ahora 'va 'bien)
```

## Manejo de números

Aunque LISP ha sido concebido especialmente para el manejo de listas, también dispone de herramientas para el proceso de números. Básicamente, vamos a considerar dos grupos de funciones para el proceso de números: algunos predicados y otras funciones aritméticas.

Un predicado en LISP es una función que a partir de sus argumentos (normalmente uno o dos) devuelve uno solo de dos posibles valores: verdadero o falso. Se representa el verdadero por «T» (True, es verdadero en inglés); el valor falso está representado por NIL (como la lista vacía).

En cada versión LISP estarán disponibles unos u otros de los predicados y funciones usuales: aquí presentaremos los más utilizados, pero si en el LISP que usted tiene a mano no están incluidos podrá definirlos después fácilmente.

También se utilizan predicados en el manejo de listas e incluso algunos predicados (como EQUAL, que se ve a continuación) son válidos para símbolos y números. Ahora presentamos algunos predicados que se utilizan con frecuencia en el proceso de números y más adelante se describirá el resto. La mayoría de los predicados tienen una «P» al final del nombre (ZEROP, NUMBERP, LESSP...), pero no es imprescindible.

EQUAL es un predicado muy utilizado (a veces aparece simplemente como EQ): significa «igual» y devolverá T (true, cierto) si los dos argumentos que se le dan son iguales; en caso contrario, devolverá NIL (falso). De este modo:

```
(EQUAL 2 2)
```

dará valor T así como:

```
(EQUAL 'A 'A)
```

e, incluso:

```
((EQUAL X 'A)
```

si previamente hemos asignado a «X» el valor «A» (ya veremos cómo se asignan valores a las variables). Por el contrario, serán NIL las siguientes listas:

```
(EQUAL 'A (A))
(EQUAL X (X)), etc.
```

En algunos dialectos se dispone de varias versiones de la igualdad según se trate de listas o números e, incluso, en alguno existe una pequeña diferencia entre EQ y EQUAL (en contra de lo que hemos dicho antes) respecto a la ubicación en memoria de los objetos o a su representación interna o a su concepción formal..., pero estos detalles, por ahora, no le deben preocupar al lector. Hay dialectos de LISP que admiten el signo de igualdad (=) en vez de EQUAL cuando se están comparando números.

Otro predicado interesante es LESSP (menor que), que comprueba si

el primer número que se le da es menor (devuelve T) o no (devuelve NIL) que el segundo número. Serán T las expresiones:

```
(LESSP 2 3)
(LESSP -5 5)
(LESSP X Y)
```

si X tiene valor numérico menor que Y.

También disponemos de GREATERP (mayor que) para comprobar la condición contraria. Así serán verdad (T):

```
(GREATERP 3 2)
(GREATERP 5 -1)
(GREATERP Y X)
```

en el mismo supuesto anterior.

En ocasiones es útil comprobar los valores individuales de alguna variable. Suele disponerse de varios predicados con esta finalidad:

- ZEROP comprueba si el valor que se le da como argumento vale exactamente cero.
- ONEP comprueba si vale «1» precisamente la constante o variable que se le da.
- ODDP será verdad si el número dado es impar (ODD en inglés); en caso contrario valdrá NIL.

Vamos a anotar, por último, que existe un predicado especial, «NOT», para cambiar de valor al predicado al que se aplica. Es decir, si la expresión que se pone de argumento de NOT es T, el resultado será NIL; si, por el contrario, la expresión a la que se aplica NOT vale NIL, el conjunto tendrá un valor de T. Así, serán T las siguientes expresiones:

```
(NOT NIL)
(NOT (NOT T))
(NOT (ODDP 2))
(NOT (NOT 12))
```

pues (NOT 12) es NIL, al ser 12 un número y no un predicado que valga NIL (normalmente, NOT dará NIL con cualquier átomo que no valga precisamente NIL —o dará error—).

Pero para el manejo de números disponemos de funciones primitivas más sofisticadas que nos permiten realizar numerosas operaciones aritméticas:

PLUS (‹‹mas››) produce la suma de los valores que se le dan como argumentos:

```
(EQUAL (PLUS 3 2) 5)
```

será T, pues (PLUS 3 2) da como resultado 5. A veces en vez de PLUS se utiliza el signo de la suma (+). Algunos dialectos admiten la suma de varios números, y no exactamente dos. Así, en un dialecto que admite estas variantes, tendrá valor T la siguiente expresión:

```
(EQUAL (+6 -2 5 3) 12) o incluso
```

```
( = (+6 -2 5 3) 12)
```

DIFFERENCE devuelve como resultado la diferencia de los dos argumentos que se le dan:

```
(DIFFERENCE -2 -6)
      4
(EQUAL (DIFFERENCE 5 6) -1)
      T
```

TIMES (‹‹veces››) devuelve el producto de los números. Así ‹‹TIMES 3 4›› habrá que leerlo como ‹‹3 veces 4››, es decir, 3 × 4. En algunas versiones de LISP se aceptan productos múltiples; en otras se usa el asterisco (*) como sím-

bolo del producto, en vez de **TIMES** (a veces la función «MUL»).

```
(TIMES (TIMES 3 2) 3)
      18
(PLUS (TIMES 6 -1) 6)
      0
```

**QUOTIENT** da el cociente de la división. Es muy normal que QUOTIENT dé el cociente entero de la división (si el **LISP** de que disponemos sólo maneja enteros, es imprescindible). En este caso se puede utilizar otra función para obtener el resto.

```
(QUOTIENT 7 5)
      1
(TIMES (QUOTIENT 9 4) 3)
      6
```

**REMAINDER** da, precisamente, el resto de la división entera de los dos argumentos que se le ponen.

```
(REMAINDER 7 5)
      2
(TIMES (REMAINDER 9 4) 3)
      3
```

En vez de **QUOTIENT** se utiliza, a veces, «**DIV**» o bien «/» y la división puede ser múltiple, dividiéndose el primer número por el segundo, el cociente entre el tercero, el resultado entre el cuarto, etc. Respecto del resto, hay dialectos en que se da el resto por defecto siempre, otros en que se da el menor (por defecto o por exceso), etc. Para ver estos detalles y otros muchos que irán apareciendo (así como la nomenclatura exacta (**DIV**, **QUOTIENT**, /, etc.) deberá consultar el manual del **LISP** que esté usted utilizando en su ordenador.

Hay otro conjunto de funciones que suele estar disponible en los diferentes dialectos de LISP (aunque aquí la variedad es mayor) y que son sumamente útiles. Comentemos algunas de las más usuales:

ABS da el valor absoluto de un número:

```
(ABS -5)
5
```

ADD1 devuelve un número una unidad mayor que el que se da como parámetro:

```
(ADD1 -3)
-2
```

(a veces, la función ADD1 aparece como «1+»)

SUB1 devuelve el número resultante de restar una unidad al argumento:

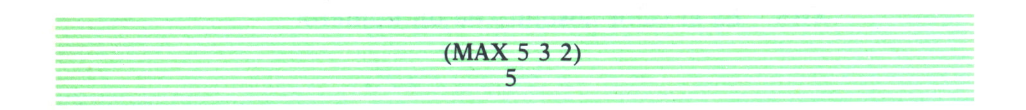
```
(SUB1 (TIMES 3 2))
5
```

(a veces, la función SUB1 se escribe como "1–")

MIN da como resultado el número menor de entre una lista que se le da:

```
(MIN 5 3 2)
2
```

MAX de un modo parecido a la anterior, devuelve el mayor valor:



```
(MAX 5 3 2)
5
```



## Evaluación de expresiones y asignación de valores

Antes de continuar describiendo los elementos del lenguaje, conviene detenerse a considerar cómo actúa el propio intérprete de LISP, para aclarar algunos aspectos de la sintaxis del lenguaje.

En todo lo que hemos descrito, hemos manejado sencillas expresiones encabezadas casi siempre por una función. Y la frase usual ha sido «si tecleamos tal expresión, el intérprete devolverá el valor cual», o expresiones semejantes. De hecho, esas afirmaciones reflejan el modo de actuar del intérprete de LISP, normalmente.

En efecto, el núcleo central del sistema es el evaluador. El evaluador de LISP es un «programa» que realiza un bucle básico en tres etapas: lectura, evaluación, escritura. Para cada frase LISP que se introduzca, se realiza este ciclo (salvo excepciones en que se rompe el ciclo normal). Por ello, este evaluador debe ser sumamente eficaz.

En la primera etapa (lectura), se toma la frase que se introduce y se examina para comprobar que se cumplen ciertas normas básicas de la sintaxis del lenguaje. Si se detecta alguna anomalía se da el mensaje correspondiente (el más usual es que faltan paréntesis) y se vuelve a la situación de lectura, hasta completar una frase LISP bien construida (una S-expresión).

La segunda fase, la etapa básica, es la de evaluación: el evaluador calcula (evalúa) los valores de los argumentos, llama a la función correspondiente y calcula el valor final; o sencillamente, evalúa el átomo que se le ha dado en la S-expresión. En ocasiones la función es compleja y los valores resultantes de la evaluación de una función se pasan a la siguiente o siguientes, de tal modo que la evaluación se realiza a varios niveles. A veces se producen «efectos de borde» (veremos lo que son) en la evaluación de una S-expresión.

En la tercera etapa se escribe el valor obtenido en la evaluación anterior.

Pero ¿cómo se realiza exactamente esta evaluación? La frase LISP o S-expresión puede ser un átomo o una lista, como hemos indicado.

El átomo puede ser constante o variable. Una constante se evalúa por su propio valor; es decir, un número se evalúa por el valor que representa (como era de esperar) y un símbolo especial por su valor también. Así, podemos mantener un diálogo (un poco «tonto», por lo demás) con el ordenador, de este modo:

```
? 25
25
? T
T
? NIL
NIL
? Variable
* error: variable no ligada*
?
```

(hemos supuesto que «?» es el «prompt» (aviso) del LISP y, por tanto, lo que aparece en las líneas que comienzan por «?» son los valores que tecleamos cuando el intérprete nos da el aviso de que está listo. En la línea siguiente aparece la «respuesta» de LISP: el resultado de la evaluación que ha realizado).

Se observará que cuando introducimos un identificador de valor no predefinido (una variable) que no corresponde tampoco a una función, el intérprete manda un mensaje de error (en su ordenador el mensaje de error tendrá otro formato y estará, problablemente, escrito en inglés). Si, por el contrario, hubiéramos asignado previamente un valor a Variable (la hubiéramos «ligado»), el intérprete la evaluaría al valor que tiene.

Esta asignación la podemos hacer con la función SETQ (ya veremos en detalle cómo) y el diálogo con el ordenador podría continuar:

```
? 25
25
? (SETQ Variable 12)
12
? Variable
12
```

Ahora no se produce error, sino que «Variable» se evalúa al valor con el que se ha ligado (es decir, 12).

La segunda posibilidad es que el evaluador reciba una lista. En este caso, supone que la cabeza de la lista será una función y el resto (la cola) serán sus argumentos. Por tanto, buscará entre las funciones disponibles (es decir, las predefinidas en el LISP en el que estemos trabajando, más las definidas por el usuario) la que se le solicita y la ejecuta con el valor de los argumentos. Si no existe esa función, dará un mensaje de error.

Pero para ejecutar la función, previamente ha de *evaluar* los argumentos. En la evaluación de los argumentos se aplican las mismas reglas descritas: es decir, si un argumento es una constante se evalúa en su propio valor; si es una variable, en el valor con el que se ha ligado (el valor que se le ha asignado) y si es una lista se supone que el primer átomo es una función y los restantes sus argumentos (argumentos que habrá que evaluar, de nuevo, antes de evaluar la función).

Así, por ejemplo, si introducimos la expresión:

```
(PLUS (TIMES -2 3) 4)
-2
```

el evaluador entiende que esa lista (de tres elementos) es una función y, por tanto, comprueba que tiene definida la función PLUS. A continuación evalúa sus dos argumentos. El primero es, a su vez, una lista: por tanto, comprueba que tiene definida la función TIMES, evalúa –2 (al valor –2; es una constante), evalúa 3 y calcula (TIMES –2 3), obteniendo, lógicamente, como valor del primer parámetro de PLUS, el número –6. El segundo argumento de PLUS se evalúa a cuatro. Por último, el cálculo de (PLUS –6 4) hace que la expresión global se evalúe en –2, que es el valor que devuelve el intérprete de LISP.

Si hacemos, por otro lado:

```
(CONS T NIL)
```

obtendremos la lista:

```
(T)
```

pues la lista introducida incluye en cabeza una función (CONS) y las otras dos expresiones (sus argumentos) tienen asignados los valores T y NIL, respectivamente.

Si, por el contrario, hacemos:

```
(CONS HOLA (QUE TAL))
```

el evaluador dará error, porque al ir a evaluar los argumentos de la función CONS se encuentra con la «variable» «HOLA» que no tiene asignado ningún valor. Producirá un error indicando que «HOLA» está *no ligada* (probablemente el mensaje será algo parecido a «Error: unbound variable. HOLA»). Como lo que queremos conseguir en este caso es, sencillamente, que LISP construya una lista con esas tres palabras, debemos indicarle que *no evalúe* los parámetros, sino que los tome tal como se los damos: insistimos, si no se le indica nada, LISP *siempre evalúa* las expresiones que se le dan. Para indicarle que *no evalúe* algo, se pone delante un apóstrofo «'» («quote», en inglés). De este modo, si le decimos:

```
(CONS 'HOLA (QUE TAL))
```

seguirá dando error, pues aunque no evalúe HOLA, intentará evaluar la «variable» QUE de la lista (QUE TAL). Pero si escribimos:

```
(CONS 'HOLA '(QUE TAL))
```

la respuesta del intérprete, una vez evaluada la función CONS con los parámetros dados, será:

```
(HOLA QUE TAL)
```

Si introducimos:

```
(CAR (HOLA QUE TAL))
```

dará error («función no definida») al ir a evaluar el argumento de la función CAR —(HOLA QUE TAL)— ha encontrado una lista y la cabeza de la lista —HOLA— no la tiene definida como función. Para evitar que «evalúe» el argumento que le hemos dado habrá que ponerle un apóstrofo delante. En efecto, al introducir:

```
(CAR '(HOLA QUE TAL))
```

contesta el intérprete:

```
HOLA
```

Por estas razones, si introducimos:

```
(PLUS '(TIMES -2 3) 4)
```

el evaluador dará error, ya que el primer argumento de PLUS no es numérico, puesto que se evalúa (se «no evalúa») a (TIMES –2 3) que es una lista, y no un número. Insistimos (TIMES –2 3) se evalúa a «–6», mientras que '(TIMES –2 3) se evalúa a «(TIMES –2 3)».

Usualmente en LISP se dispone de una palabra que realiza la misma tarea que el apóstrofo: es QUOTE.

En efecto:

```
(CAR ' (HOLA QUE TAL))
                                    y
(CAR (QUOTE (HOLA QUE TAL)))
```

son para LIPS equivalentes.

QUOTE no es propiamente una función: se limita a devolver el valor que se pone como argumento. En el argot de LISP se llama a esta estructura «forma especial»:

```
(QUOTE (una lista))
```

es equivalente a:

```
'(una lista).
```

Por ejemplo, si suponemos que la variable X tiene asignado un valor de «Temperatura», la expresión:

```
(SET X 25)
```

hará que la variable «Temperatura» tenga un valor de 25. Por tanto, se podría establecer el diálogo:

```
? (SET X 25)
25
```

```
? X
Temperatura
? Temperatura
25
```

(obsérvese cómo la expresión (SET X 25) además de atribuir a Temperatura el valor 25, se evalúa —ella misma— a 25, que es el valor que devuelve el intérprete de LISP). Insistimos en que, de acuerdo con las «normas de evaluación» definidas anteriormente antes de llamar a la primitiva «SET», el evaluador de LISP *evalúa* ambos argumentos (25 se evalúa a 25 y X, una variable, al valor al que esté ligada, suponemos que «Temperatura»).

Sin embargo, puede haber casos en que no nos interese evaluar el primer argumento (o ambos). Para ello, pondremos el apóstrofo (o QUOTE) delante del argumento.

Por ejemplo, para ligar la variable «X» con el valor «Temperatura» (paso previo al anterior que consistía en asignar a «Temperatura» el valor 25) debemos escribir:

```
(SET 'X 'Temperatura)
```

así LISP no evalúa ni «X» ni «Temperatura», sino que, sencillamente, «liga» la variable «X» con el valor «Temperatura». El diálogo completo podía ser:

```
? (SET 'X 'Temperatura)
Temperatura
? (SET X 25)
25
? X
Temperatura
? Temperatura
25
```

O también:

```
? (SET (QUOTE X) (QUOTE Temperatura)
Temperatura
? (SET X 25)
25
? X
Temperatura
? Temperatura
25
```

Esta combinación, consistente en utilizar la función «SET» y a continuación un apóstrofo (o QUOTE), para asignar el valor del segundo argumento (con o sin apóstrofo, según los casos) al primer operando *sin evaluar éste,* es muy común, por lo que existe una función primitiva que cumple esta tarea: es «SETQ» (equivalente a un SET con QUOTE detrás). La asignación anterior se podría, por tanto, hacer del siguiene modo:

```
? (SETQ X ' Temperatura)
Temperatura
```

y surtiría el mismo efecto que (SET (QUOTE X)...) Ya hemos utilizado esta expresión anteriormente en el diálogo:

```
? 25
25
? (SETQ Variable 12)
12
? Variable
12
```

donde se pone «SETQ» para asignar a «Variable» el valor «12», sin evaluar «Variable», pues como «Variable» no estaba ligada a ningún valor, daría error la expresión:

```
(SET Variable 12)
```

Existe una función primitiva (EVAL) que produce el efecto contrario a QUOTE: en efecto, con «quote» se inhibe la evaluación de un átomo con «EVAL» se «fuerza» dicha evaluación. Así, por ejemplo:

```
(TIMES 2 3)
```

se evalúa a 6, mientras que:

```
'(TIMES 2 3)
```

se «evalúa» a (TIMES 2 3), como hemos dicho. Por el contrario:

```
(EVAL '(TIMES 2 3))
```

se evalúa, de nuevo, a 6.

Otra función relacionada, asimismo, con el proceso de evaluación es APPLY. Esta primitiva procesa un átomo (que debe ser una función) y una lista (que deben ser los argumentos de dicha función) y «aplica» los argumentos a la función para su ejecución, *sin evaluar* previamente ni el átomo representativo de la función ni la lista de los argumentos. Así:

```
(APPLY ' TIMES '(2 3))
```

se evalúa a 6 y.

```
(APPLY 'SUB1 '(3))
```

se evalúa a 2.



## Efectos laterales

Hemos visto que, como resultado de todas las evaluaciones en LISP, el intérprete «devuelve» el valor obtenido en la evaluación pero que, además, se pueden producir otras acciones. Estas acciones son, a veces, más importantes que el propio valor devuelto: son los llamados «efectos laterales» de la evaluación.

Por ejemplo, si hacemos

```
(CAR '(A B C D))
```

el intérprete devuelve A y la obtención de este valor es el efecto principal esperado; o bien en

```
(EQUAL (CAR '(A B C D)) X)
```

que devolverá T si X está ligada previamente al valor «A» o «NIL» si no es así.

Pero hay otros casos en que el valor devuelto no es importante, sino el «efecto lateral» que se produce:

```
(SETQ Variable 12)
```

devuelve «12», pero lo que nos interesa es que «Variable» tenga asignado, a partir de ahora, el valor «12».

Otro caso curioso es aquél en que en una función se realizan varias tareas (varias evaluaciones): el intérprete devolverá el valor de la última evaluación que haya realizado, pero son interesantes el resto de los efectos laterales conseguidos. Así, si definimos (inmediatamente vamos a ver cómo se utiliza «DEFUN» para definir funciones):

```
(DEFUN MATRIZDOBLE (X Y Z)
  ((SETQ X1 (PLUS X X)) (SETQ Y1 (PLUS Y Y))
   (SETQ Z1 (PLUS Z Z)))
```

y llamamos a la función mediante la expresión:

```
(MATRIZDOBLE 2 3 6)
```

el intérprete devolverá:

```
12
```

que es el resultado de la última evaluación (Z1 es igual al doble de Z), pero lo realmente interesante es que en X1, Y1 y Z1 están los valores dobles de 2, 3 y 6.

Otra situación en la que son importantes los efectos laterales se produce en el uso de las funciones primitivas de entrada-salida (escritura en la impresora, lectura de un archivo en disco, etc.): independientemente del valor que LISP devuelva por pantalla, lo importante es la acción de entrada-salida producida.



## Definición de funciones

Como ya hemos comentado, una de las características que más valora el programador de LISP es la facilidad que existe en este lenguaje para definir funciones.

Para definir una nueva función en LISP (adicional a las primitivas que ya están definidas de origen en el lenguaje) se utiliza la «forma especial» DEFUN. En otros dialectos se usan los átomos DE, DEF, DEFINE..., con la misma finalidad. Esta forma especial es una lista con la siguiente estructura: el primer átomo es la palabra DEFUN; el segundo, el nombre de la función cuya definición estamos haciendo; a continuación una lista cuyos elementos son las variables que se van a utilizar en la definición; por último, el «cuerpo» de la definición (es decir, la «descripción» de la función en lenguaje LISP). Por ejemplo, si en nuestro LISP no está definida como primitiva, la función ADD1 descrita anteriormente la podríamos definir así:

```
(DEFUN ADD1 (X) (PLUS X 1))
```

Con ello LISP almacena el dato de que (ADD1 X) es lo mismo que (PLUS X 1). Del mismo modo podríamos definir:

```
(DEFUN CUBO (X) (TIMES X X X))
```

o, si el intérprete no admite productos múltiples,

```
(DEFUN CUBO (X) (TIMES (TIMES X X) X))
```

o, manejando listas,

```
(DEFUN SALUDO (X) (LIST 'HOLA 'QUE 'TAL X))
```

Si ahora escribimos:

```
(SALUDO 'PEDRO)
```

el intérprete devolverá:

```
(HOLA QUE TAL PEDRO)
```

o bien, si previamente teníamos asignado a X el valor «Pedro» (bien por alguna evaluación anterior o como resultado de una SETQ), se puede hacer, con el mismo resultado,

```
(SALUDO X)
```

Nótese que en la definición de la función las variables deben ir entre paréntesis (formando una lista) y sin apóstrofos. A su vez, en el cuerpo de la definición las variables no deben ir entre paréntesis ni con apóstrofos (aunque deben llevar apóstrofos los átomos que no queramos que se evalúen al «llamar» a la función).

Aunque en la definición de la función las variables quedan sin ligar, cuando se llama a la función, hay que dar valores a los argumentos y entonces las variables de la definición se «ligan» a estos valores (temporalmente; al concluir la evaluación de la función, se liberan de nuevo).

Si queremos en cualquier momento ver cómo está almacenada una función que hayamos definido, podemos introducir el nombre de la función sin parámetros y obtendremos (en los dialectos más usuales, al menos) una definición del siguiente estilo:

```
(LAMBDA (X) (LIST ('HOLA 'QUE 'TAL X))
```

que, como se ve, es la definición de la función sin su nombre: en efecto, como definición de la palabra representativa de la función, LISP almacena esta «lambda expresión». Una «lambda expresión» es una lista en la que aparecen los parámetros y una expresión LISP a evaluar con esos parámetros. Podríamos haber descrito anteriormente las «lambda expresiones» y haber dicho después que en la forma especial DEFUN se da un nombre a una «lambda expresión».

Pero las «lambda expresiones» se pueden utilizar como tales dentro de un programa LISP cuando hay que realizar una labor repetidas veces y queremos definir localmente una pequeña función para ejecutar esa labor: como sólo se va a usar en un lugar del programa no es necesario darle nombre y se usa una «lambda expresión» que es, en el fondo, como una función sin nombre, para ser usada localmente. (Piénsese que en programas largos puede ser útil esta economía de nombres para evitar duplicidades y conseguir un pequeño ahorro de espacio.)



## Control del proceso

Para la toma de decisiones dentro del programa se pueden utilizar varias «formas especiales» (es decir, una especie de función con estructura peculiar que *no evalúa* sus argumentos).

Dependiendo del dialecto de LISP que esté usted utilizando dispondrá de unas funciones u otras. Vamos a comentar aquí las más usuales, aun-

que en su LISP existan otras a las que no aludiremos. Por supuesto, ya hemos visto de qué forma sencilla se pueden definir funciones en LISP y, por tanto, usted podrá incluir estas funciones en su intérprete, si no las tiene ya disponibles.

**IF**

La condicional más sencilla de LISP es IF. Es una forma especial formada por una lista que tiene cuatro elementos: el primer átomo de la lista es la palabra IF; el segundo una expresión LISP que se evalúa para establecer el control (es la «condición» del IF); a continuación aparecerán dos expresiones que son las que se evaluarán respectivamente si la condición se cumple o si no se cumple. Por ejemplo,

```
(IF (EQUAL X 1) '(VALE UNO) '(NO ES UNO))
```

devolverá (VALE UNO) si X tiene asignado un valor de «1» y devolverá (NO ES UNO) si X tiene asignado un valor distinto de 1.

Se puede utilizar IF para definir, por ejemplo, una función (un predicado) que averigüe si un número es cero o distinto de cero, así:

```
(DEFUN CERO (X) (IF (EQUAL X 0) T NIL))
```

Una vez introducida la expresión anterior, si tecleamos:

```
(CERO (SUB1 1))
```

obtendremos como respuesta:

```
T
```

Si, por el contrario, introducimos:

```
(CERO (ADD1 1))
```

LISP devolverá:

```
NIL
```

## COND

Otra forma especial muy útil para la toma de decisiones en un programa LISP es el condicional COND. Tiene la estructura de una lista cuyo primer átomo es, precisamente, COND y los restantes elementos (tantos cuantos se quiera poner) son parejas de términos (entre paréntesis) formados por dos elementos: el primero un predicado y el segundo una expresión LISP. Es decir, su estructura es:

```
(COND
   (predicado1 expresión1)
   (predicado2 expresión2)
   (predicado3 expresión3)
             »
             »
             »
   (predicadon expresiónn))
```

Para procesar esta expresión condicional, el intérprete va evaluando sucesiva y ordenadamente los predicados hasta que encuentra uno que valga T (True, verdad); entonces evalúa la expresión correspondiente, devuelve ese valor y concluye el proceso. Sólo evalúa la expresión correspondiente al *primer predicado* verdadero en la lista.

Las expresiones de la derecha de las parejas anteriores pueden estar formadas por toda una sucesión de expresiones (y no necesariamente sólo una): en ese caso se evaluarán todas las que aparecen a la derecha del pri-

mer predicado verdad, se ejecutarán (si es que tienen algún efecto lateral) y se devolverá el último valor evaluado, como siempre (o casi siempre).

Naturalmente, si se introduce en COND una sublista tal que la cabeza de ella es un predicado que siempre se evalúa a «T», la evaluación no seguirá jamás después de esta sublista, por lo que se puede omitir el resto de las sublistas que estén detrás. A veces se busca precisamente este efecto: hay ocasiones en que es necesario programar un condicional del tipo «si A1, entonces M1; si A2, entonces M2..., si A5, entonces M5 y, en caso contrario, M6». Esto se escribirá en LISP con una estructura como la siguiente:

```
(COND (A1 M1)
      (A2 M2)
        »
        »
        »
      (A5 M5)
      (T M6))
```

Si alguno de los predicados A1 hasta A5 es «T», LISP evaluará la secuencia correspondiente M1, M2..., M5 y, si ninguno es «T», ejecutará M6 (pues «T» siempre es «T», claro).

## AND

Esta es una función primitiva utilizada para establecer condiciones complejas. Significa «y».

La función «AND» se evalúa del siguiente modo: se van tomando sucesivamente cada uno de sus argumentos y se evalúan; si uno de ellos devuelve NIL, entonces se concluye la evaluación y la función AND devuelve NIL. Si ninguno devuelve NIL, AND devuelve el último argumento evaluado. Esto puede producir sorpresas cuando se evalúan valores que no son predicados (funciones, variables, etc.) y que, por tanto, no devuelven ni «T» ni «NIL». Así, el resultado de

```
(AND 'HOLA 'QUE 'TAL)
```

será

```
TAL
```

Si sabemos que X vale «1» e Y vale «3» y escribimos:

```
(COND ((AND (EQUAL X 1) (LESSP Y 5)) T) (T NIL))
```

LISP devolverá «T», pues para evaluar COND toma la primera sublista y dentro de ella la cabeza: (AND (EQUAL X 1) (LESSP Y 5)); si suponemos que «X» estaba ligado al valor 1 e «Y» a un valor menor que 5, como se cumplen ambas condiciones («una condición AND —Y— la otra») entonces, la cabeza de la lista se evalúa a «T» y se devuelve el resultado de evaluar la cola de dicha sublista: «T». (De acuerdo con las condiciones de evaluación de COND, no pasa a evaluar la segunda sublista (T NIL).)

**OR**

A semejanza de la anterior, OR se utiliza en la construcción de cláusulas de condición complejas. Hay que interpretarla como una disyunción («o»). Se evalúa de tal modo que si cada argumento va dando valor NIL, sigue la evaluación: al final se devuelve valor NIL. Si, por el contrario, alguno de los argumentos devuelve un valor distinto de NIL (un valor alfanumérico o bien «T» si se trata de predicados), cesa la evaluación y se devuelve el último valor evaluado. Por ejemplo,

```
(OR (EQUAL 5 1) (LESSP 2 5))
```

dará valor «T», pues es ése el valor que se obtiene en la evaluación de (LESSP 2 5). Pero,

```
(OR (EQUAL 5 1) (CDR '(A B C)))
```

dará una respuesta de

```
(B C)
```

**NOT**

También se puede incluir en las expresiones condicionales complejas el predicado NOT, del que ya hemos hablado.

Por otro lado, normalmente (depende de los dialectos) no hay problema en establecer condiciones complejas con todas estas primitivas condicionales mezcladas (o, incluso, apareciendo alguna de ellas varias veces). Así, por ejemplo, la respuesta a

```
(OR (AND (EQUAL 5 1) (LESSP 2 5)) (NOT EQUAL 3 3)))
```

será NIL.

Por otro lado, existen varias primitivas que permiten establecer una cierta estructuración permanente del programa. Básicamente se pueden reseñar dos grupos: uno primero formado por las primitivas básicas de establecimiento de bucles (suele haber, en cada dialecto de LISP, al menos una de ellas) y son las primitivas LOOP («lazo» o «bucle»), REPEAT («repite»), DO («haz»), etc.; otro grupo es el que incluye a WHILE («mientras»), UNTIL («hasta») y equivalentes, que controlan el final del bucle.

Las primitivas LOOP, REPEAT, etc., proporcionan un procedimiento de establecer un bucle: aunque la sintaxis concreta varía con cada dialecto, normalmente suelen tener después del nombre una condición (y una asignación de variables locales en el caso de «DO») y a continuación un conjunto de expresiones a evaluar repetitivamente con los valores de las variables, hasta que se cumpla la condición de terminación indicada al principio.

Esta condición se suele establecer con las primitivas WHILE («mientras») o UNTIL («hasta»), que hacen que el proceso se repita «mientras» se cumpla una determinada condición o precisamente «hasta» que esa condición se dé.



# REFERENCIA ADICIONAL DE PRIMITIVAS

Aunque con las primitivas ya descritas se pueden preparar pequeños programas en LISP, la capacidad del lenguaje es mucho mayor. Entendemos que rebasa los límites de este libro la discusión de las técnicas sofisticadas de programación en LISP y, además, para profundizar más en algunos aspectos de dichas técnicas deberíamos referirnos a alguno (o algunos) de los numerosos dialectos existentes (algunos no disponibles en ordenadores personales). Creemos, sin embargo, que puede ser útil para completar el panorama que hemos presentado del lenguaje, a modo de introducción a la programación en LISP, la descripción de algunas de las primitivas más usuales no comentadas aún, insistiendo en que los detalles concretos de la sintaxis en el dialecto o versión de LISP de que usted disponga, deberá consultarlos en el manual concreto que le ha facilitado el fabricante del intérprete correspondiente.



## Otros predicados interesantes

Suelen existir implementados en cualquier intérprete de LISP varios predicados (aparte de los ya reseñados) sumamente útiles.

**NULL**

Comprueba si el argumento (sólo uno) que se le da es NIL o no.
Por tanto, devolverán valor T las siguientes expresiones:

```
(NULL ())
(NULL (CDR 'HOLA))
```

**ATOM**

Devolverá valor T si el objeto que se le pone como argumento (uno solo) es un átomo. Por ejemplo,

```
(ATOM HOLA)
(ATOM (CAR '(HOLA QUE TAL)))
```

## LISTP

Averigua si el objeto que tiene como argumento es una lista. Es decir, será T:

```
(LISTP (CONS 'HOLA 'PEPE))
```

## NUMBERP

Comprueba el tipo del objeto y devuelve T si es un número:

```
(NUMBERP 12)
```

## BOUNDP

Devuelve el valor T si la variable que se le da está ligada («bounded» en inglés). Por ejemplo,

```
? (BOUNDP 'PEPE)
  NIL
? (SETQ PEPE 'MAYOR)
  MAYOR
? (BOUNDP 'PEPE)
  T
```

## MAKUNBOUND

No es propiamente un predicado, sino una función que hace que la variable que se da como argumento deje de estar ligada («making unbound», en inglés). No todos los dialectos disponen de esta primitiva.

## SYMBOLP

Devuelve T si el argumento que se le da es, precisamente, un símbolo (y no un número).



# Primitivas de manejo de listas

### ASSOC

Se utiliza para localizar datos en una lista «larga» (una base de datos). La base de datos debe tener la estructura de sublistas y estar formada cada una de estas sublistas por dos elementos: la cabeza de la lista como identificador y el resto de la lista (cola) como conjunto de datos. Si se asocia (SETQ) la base de datos a un identificador (sea este «DATOS», por ejemplo) la introducción de

```
(ASSOC 'identificador DATOS)
```

buscará en la lista de DATOS (la «base de datos») una sublista cuya cabeza sea el «identificador» y devolverá el resto de la sublista como una lista.

### NTH

Es un selector que extrae el n-simo elemento de una lista. (A veces extrae el CDR de la lista desde el n-simo elemento; vea el manual del dialecto LISP que esté usted utilizando.)

### NTHCDR

Es una función que devuelve el n-simo CDR de la lista que se le da. Tal como acabamos de comentar, hay dialectos en que esta función es la realizada por NTH. En cualquier caso, la combinación de la primitiva adecuada (NTH o NTHCDR) con CAR o CDR (según los casos) permite ir extrayendo elementos de la lista.

### REVERSE

Devuelve la lista «inversa» de la lista que se le da. Es decir, se le da una lista y la devuelve con los elementos ordenados de un modo inverso a la lista original.

## APPEND

Reúne en una sola lista los elementos de dos listas distintas. En algunos dialectos se pueden reunir, en una sola operación de «append», varias listas.

## LENGTH

Devuelve la longitud de la lista que se le da. Hay varios dialectos que tienen esta función bajo el nombre LEN.

## EXPLODE

Produce una lista con los diferentes caracteres de un átomo. Así, si se introduce

```
(EXPLODE 'PEPE)     (PEPE es un átomo)
```

LISP devolverá

```
(P E P E)           ( (P E P E ) es una lista )
```

## IMPLODE

Devuelve un átomo reuniendo todos los elementos (átomos) de una lista. Es la función inversa de la función EXPLODE. Así, el resultado de

```
(IMPLODE '(P E P E))
```

será

```
PEPE
```



## Primitivas de entrada/salida

Para las funciones de entrada y salida suele estar disponible en LISP un conjunto de primitivas muy variado: en efecto, estas funciones son más dependientes aún del equipo y de la versión de intérprete utilizada que el resto de funciones a que hemos aludido.

Para la impresión de datos suele estar presente la primitiva PRINT y algunas variantes; y para la introducción de datos la función READ. Es muy normal tener que «abrir» (OPEN) un dispositivo antes de leer o escribir en él, para «cerrar» (CLOSE) al concluir la lectura o escritura de datos.

Además, para guardar los programas que hayamos podido preparar, las asignaciones de variables realizadas, etc., tendremos, normalmente, que utilizar la primitiva SAVE o similar; y para «cargar» en memoria un programa y el estado del área temporal en que el programa estaba operando, se suele disponer de la función LOAD.



# TIPOS Y DIALECTOS DE LISP

Teniendo en cuenta que LISP es uno de los más antiguos lenguajes de alto nivel (junto con FORTRAN), no es de extrañar que existan numerosas versiones y dialectos del lenguaje. Además, hasta hace una década tuvo una extensión muy limitada estando reducida a ciertos ambientes universitarios y de «laboratorios» de investigación en informática. Sólo modernamente, con el desarrollo de las técnicas de Inteligencia Artificial, ha vuelto a tener interés comercial el lenguaje y se ha abordado el problema de la compatibilidad y transportabilidad entre los «diferentes LISP's» existentes.

Por otro lado, la compatibilidad se complica con la existencia de «máquinas LISP»: es decir, no sólo existen numerosas versiones para que se utilicen sobre diferentes ordenadores de propósito general, sino que se han desarrollado máquinas concebidas para ser programadas en LISP: esto significa que se desciende a nivel de microcódigo para aprovechar hasta el extremo las capacidades de la propia máquina y del lenguaje, pero esto significa, por otro lado, que se introducen «peculiaridades» útiles para optimizar las características del lenguaje que se obtiene.

En el último lustro las empresas que tenían desarrollados intérpretes de LISP, o iban a desarrollarlos, han enfrentado el problema de la portabilidad de sus intérpretes, por lo que han proliferado las versiones de LISP escritas en C y otros lenguajes evolucionados.

Todo ello ha producido un abanico enorme de dialectos y versiones de LISP, así como «sistemas LISP» formados por la integración de los intérpretes de LISP con otras «herramientas» de programación como intérpre-

tes de PROLOG u otros lenguajes y manejadores de los entornos adecuados. Dentro de este mundo de dialectos y versiones se ha intentado (bajo directrices del Departamento de Defensa de los EE.UU., verdadera «locomotora» y financiador de numerosos proyectos en este área) construir un estándar: así ha nacido el CommonLISP pero, hasta el presente, está lejos de conseguirse esa unificación de los dialectos de LISP.

Lo que sí es cierto es que la estructura básica del lenguaje se mantiene y hasta un cierto «estilo de programación» LISP. Al pasar de un dialecto a otro cambian funciones básicas (desde DEFUN hasta PROG, etc.) e, incluso, cosas tan elementales como la respuesta de las primitivas CAR o CDR al átomo NIL. Pero, por otro lado, cualquier persona que conozca la programación básica en LISP (en cualquier LISP no muy específico) puede, a la vista del manual de referencia del suministrador del software, detectar las diferencias existentes y adaptar su programación al dialecto disponible (otra cosa son los programas ya hechos para otro dialecto).

En lo que sigue vamos a comentar algunos de los dialectos LISP disponibles hoy en el mercado, fijándonos básicamente (aunque no exclusivamente) en los que se utilizan en las máquinas de propósito general, no muy grandes y ordenadores personales, que son los que más interesarán a los lectores de este libro.

## MAC LISP

Una de la primeras versiones LISP comercializadas extensamente fue el MacLISP del PDP-6. Esta versión fue escrita en el MIT ( «Massachusetts Institute of Technology») en la mitad de los años sesenta. Posteriormente, en la Universidad de Stanford se mejoró y aumentó la versión de MacLISP de que disponían en su PDP-6, creando una versión más adaptada al entorno DEC, que nombraron LISP 1.6 y que fue conocida, también, como el LISP de Stanford.

En la Universidad de California (en Irvine) prepararon una modificación de este «Stanford LISP» y lo bautizaron con su nombre: UCI, LISP. Allí mismo (en UCI) se desarrolló posteriormente el BBN LISP, que es el mismo UCI LISP mejorado con la inclusión de varias «herramientas» de programación (paquete de edición, paquete de desarrollo...). Este es el que, posteriormente mejorado se conoce como InterLISP.

Sin embargo, por su cuenta, siguió desarrollándose MacLISP, y ya en 1972 John L. White lo reescribió para PDP-10 e, incluso, en los años ochenta, está en uso en numerosas instalaciones de DEC-20. Una característica muy interesante de MacLISP es que cuenta con un amplio sistema «runtime» (es decir, no para desarrollo, sino para ejecución final) que incluye un intérprete escrito en Ensamblador. Este sistema «runtime» es grande (su listado ocupa unas 600 páginas) y, por tanto, eficaz.

Una de las características más sobresalientes de MacLISP es su eficacia en el manejo de números y operaciones aritméticas. En MacLISP para imprimir una función se usa la primitiva GRINDEF: úsela si usted trabaja en MacLISP y duda cómo está definida alguna función. MacLISP ha sido un clásico entre los dialectos de LISP y, de hecho, ha sido durante muchos años el punto de referencia para otros dialectos en cuanto a compatibilidad, rapidez de ejecución, facilidades, etc. Además, ha sido sobre MacLISP como se han desarrollado otros dialectos como NIL, Spice Lisp, Zetalisp, S-1 Lisp, Frazlisp, etc.

La sintaxis de Lisp-Machine LISP y de CommonLISP es semejante a ésta (excepto que CommonLISP no acepta FEXPR).

## ZETALISP

Es un dialecto obtenido a partir de Lisp-Machine LISP del MIT. Este último dialecto es el resultado del proyecto del MIT sobre la «Lisp Machine». Posteriormente han sido dos compañías comerciales (Symbolics Inc. y Lisp Machine Inc. —LMI—) quienes han comercializado dos versiones de la máquina CADR desarrollada en el MIT en el proyecto a que estamos aludiendo. Pero tanto, estas versiones comerciales (LMI CADR y Symbolics LM2) como la máquina LISP del MIT (MIT CADR) ruedan el mismo LISP, que actualmente es el ZetaLISP.

Sus características básicas vienen definidas por las máquinas en que está soportado: hacen un amplio uso de microcódigo, poseen un direccionamiento en memoria virtual a dos niveles, disponen de una memoria extensa, etc. Es especialmente eficaz en la llamada de funciones (por el sofisticado sistema de llamadas invertidas, de que ha sido dotado) y en el manejo de números en coma flotante.

## FRANZ LISP

Fue escrito por Richard Fateman y su equipo en la Universidad de California (Berkeley). Aunque en principio fue desarrollado con la finalidad concreta de hacer ejecutable sobre VAX una versión específica del sistema MACSYMA (Sistema experto utilizado en el aprendizaje de las matemáticas), ha llegado a ser una de las versiones de LISP más disponible sobre máquinas que operen bajo UNIX.

En FRANZ LISP las primitivas están escritas en minúsculas. La definición de funciones se hace, como en MacLISP, mediante la primitiva DEFUN («defun» en FRANZ LISP), aunque dispone de otra forma especial, llamada «def». El listado de una función se hace mediante «pp». El núcleo de Franz LISP está escrito casi enteramente en lenguaje C. Es bastante flexi-

ble en cuanto al soporte de arrays y estructuras definidas por el usuario, aunque no admite la programación orientada al objeto ni la definición de «tipos abstractos de datos».

## INTERLISP

InterLISP es una versión ampliada del BBN LISP desarrollado en la Universidad de California (Irvine). Básicamente, las mejoras vienen en el sentido de incluir cantidad de módulos de utilidad que ayudan en la programación, en la edición, en el manejo de archivos, etc. En InterLISP no se manejan las «macros» en el sentido usual de este término, sino que forman parte del programa solamente en el momento de la compilación. El listado de funciones se hace en InterLISP mediante la primitiva «pp».

La versión más conocida de este dialecto es la ofrecida por XEROX: InterLISP-D. Lo que aporta, básicamente, InterLISP-D es un entorno, sofisticado y sumamente cómodo, de programación: las funciones del usuario pueden pasar al entorno y viceversa; las herramientas de desarrollo son numerosas: se manejan muy cómodamente los ficheros de programas, desde ventanas; hay editores, un «inspector» (para el examen de estructuras complejas de datos), un «masterscope» (para analizar los programas del usuario), etc. XEROX propone con su InterLISP-D la realización de lo que Sheil (1983) introdujo como «programación por exploración»: es decir, el ir desarrollando el programa definitivo mediante aproximaciones sucesivas, añadido de otros elementos, reforma de los existentes, etc.

## COMMON LISP

Tal como se ha comentado, intenta ser un estándar de LISP y para los equipos no muy especializados (ordenadores personales y estaciones de trabajo utilizadas en entornos de programación de Inteligencia Artificial) lo va consiguiendo.

Existen numerosas implementaciones de él, y como la mayoría de ellas aportan alguna característica especial (aparte de la implementación del COMMON LISP en sí) se suelen nombrar dándole «su apellido»: VAX Common LISP (que rueda en todos los modelos VAX de Digital E.C. bajo el sistema operativo VMS), Data General Common LISP (que rueda en la gama completa de los ordenadores de esa firma con arquitectura MV y bajo los sistemas operativos AOS/VS y MV/UX)... y el más popular, Golden Common LISP (desarrollado por Golden Hill para los ordenadores PC de IBM o compatibles, bajo sistema operativo MS/DOS).

Este Golden Common LISP entendemos que es el más interesante para los usuarios de ordenadores PC compatibles, pues es el más extendido a

este nivel y aporta un conjunto de «facilidades» adicionales muy útiles: el editor GMACS (entorno de desarrollo con múltiples buffers, ventanas, identificación automática del código, etc.), un sistema de ayuda en línea (utilísimo en desarrollo), facilidades de gráficos, etc.

En esta misma línea de intérpretes de LISP no muy sofisticados (aunque suficientemente potentes), para su uso en ordenadores personales, podemos reseñar los siguientes:

**MULISP**

Aporta una buena capacidad para la definición (por parte del usuario) de estructuras de datos complejas (maneja tres objetos de datos primitivos: nombres, números y nodos) y dispone de numerosas primitivas para la eficaz utilización de estas estructuras. Se presenta con numerosos ejemplos y un curso «tutorial» para el fácil aprendizaje.

**BYSOLISP**

Es compatible con MacLISP y con LISP 1.5. Maneja archivos secuenciales, pero no de acceso directo. Maneja listas, símbolos (átomos) y algunos tipos de números. También maneja estructuras de datos definidas por el usuario. Es un intérprete que se facilita sin protección y a un coste no muy elevado. Aunque no es muy potente y la documentación que aporta es un poco pobre, es una magnífica herramienta para empezar a programar en LISP.

**WALTZ LISP**

Está basado en Franz LISP. Es bastante compatible con la mayoría de los dialectos de LISP que ruedan bajo sistema operativo UNIX. La mayor deficiencia de Waltz LISP es que no maneja números en coma flotante, aunque soporta listas y símbolos (átomos). Maneja archivos secuenciales y también archivos de acceso aleatorio y todas las funciones de directorios del PC-DOS. Es bastante rápido e incluye una versión de PROLOG (Clog Prolog) un poco lenta, pero interesante. Se facilita una buena documentación.

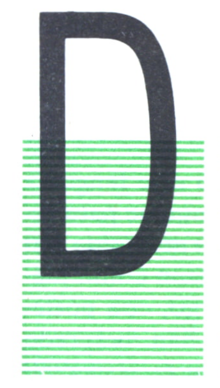
**XLISP**

Es una versión muy útil de LISP (sus autores la definen aún como «experimental») que combina las capacidades básicas del lenguaje con las ca-

racterísticas necesarias para realizar programación orientada al objeto (están definidas en XLISP las primitivas «Object» —objeto— y «Class» —clase— con esta finalidad). Está escrito completamente en C y se facilita el lenguaje fuente para que pueda ser ampliado o adaptado por el usuario. La documentación es un poco pobre, pero suficiente. Aunque las primeras versiones estaban basadas en MacLISP (sin llegar a ser compatible), en la versión 1.4 ya se introducen algunas funciones de Common LISP y los autores manifiestan su intención de llevar XLISP a ser compatible con Common LISP, en versiones futuras.

## PRESENTACION DE PROLOG





DE entre los lenguajes hoy disponibles para aplicaciones de Inteligencia Artificial es necesario destacar el PROLOG como una de las herramientas fundamentales en los desarrollos prácticos. Existen numerosas versiones de este lenguaje y hasta las compañías que defendían hace cinco años el uso de LISP (a veces en exclusiva), están incorporando hoy a sus máquinas y entornos de programación el PROLOG.

Hay que señalar dos datos básicos que marcan el enorme desarrollo que está teniendo el lenguaje PROLOG en los últimos años: por un lado, es un lenguaje muy apropiado para los sistemas de inferencia que están en la base de la mayoría de las aplicaciones de Inteligencia Artificial; de hecho, los dos lenguajes de programación más utilizados en este área son LISP y PROLOG, y los entornos más sofisticados que se están desarrollando en los últimos años se basan en alguno (o ambos) de estos dos lenguajes. Por otro lado, el PROLOG ha sido tomado como base del (de los) lenguaje(s) de programación a desarrollar en el proyecto japonés «del ordenador de la quinta generación».

¿Quiere esto decir que PROLOG va a ser el lenguaje de programación del año 2000? Esto es difícil de contestar, pero lo que parece claro es que PROLOG o algún dialecto de él será un lenguaje básico en los ordenadores que en esa fecha estemos utilizando.

Y ¿por qué PROLOG precisamente y no otro lenguaje de programación? Ya hemos comentado que no es sólo PROLOG el lenguaje a utilizar en Inteligencia Artificial, sino uno de los posibles. Todos ellos tienen una característica especial: son lenguajes declarativos y no procedurales. Esto, como veremos con detalle más adelante, supone varias ventajas prácticas, además de la facilidad general que aporta para la representación del conocimiento y el proceso de la información en tareas de inferencia; estas ventajas son que cada porción de un programa (cada proposición o con-

junto de ellas) es evaluable independientemente del resto y, por otro lado, que las modificaciones son fáciles de realizar, precisamente por esta independencia de cada proposición.

El concepto de lenguaje declarativo (en contraposición a los lenguajes procedurales) responde a la idea de facilitar al ordenador la información de que disponemos (algo estructurada, claro está) y dejar que él la maneje para obtener los resultados que deseamos, en vez de tener que concebir nosotros el procedimiento de cálculo (algoritmo) y explicárselo al ordenador, como se hace en los lenguajes convencionales.

En concreto en PROLOG, lo que realizamos es «programación lógica» (PROLOG = PROgraming LOGic): un programa en PROLOG es un conjunto de proposiciones lógicas que el intérprete es capaz de manejar. PROLOG supone, de hecho, un conjunto de tres elementos: un lenguaje de representación del conocimiento, una máquina de deducción y una verdadera metodología de programación.

Es cierto, en efecto, que en un principio fue concebido este lenguaje por Alain Colmerauer y Philippe Roussel (1972) como un lenguaje para representar los razonamientos lógicos e implementar el recientemente obtenido «principio de resolución» (propuesto por Alan Robinson como método de demostración automática de teoremas). Sin embargo, con posterioridad se ha desarrollado enormemente el lenguaje y se han mejorado sus prestaciones para su utilización en aplicaciones prácticas (especialmente sistemas expertos y procesamiento del lenguaje natural), conservando su estructura general y sus cualidades.

La programación en PROLOG ha pasado de ser una mera «programación lógica» a ser una programación con una estructura lógica de representación y proceso de los conocimientos: el modo de representar las informaciones en un programa escrito en PROLOG es una manera «más lógica» y «más natural» y marca, en consecuencia, todo un estilo de programación y una verdadera metodología de concebir y plasmar los problemas para su óptima resolución.

Pero en PROLOG, además de tener un sistema de representación del conocimiento muy cómodo y eficaz y una estructura de programación muy próxima al pensamiento humano, tenemos una verdadera máquina de deducción; en el propio intérprete se incluye un sistema de inferencia lógica que permite obtener datos y sacar conclusiones nada más formular nuestro conocimiento en aseveraciones y relaciones.

Por ejemplo, podemos decirle que Carlos I fue el padre de Felipe II y que Felipe el Hermoso fue padre de Carlos I; si, además, le decimos que el «padre del padre de alguien» es el abuelo de esa persona, podemos inmediatamente pasar a preguntarle si Carlos I es padre de alguien, si Felipe el Hermoso es abuelo de alguien, quién es el padre de Felipe II, etc., y el propio sistema hará sus «cálculos» y nos contestará. (Por el contrario, en un lenguaje más convencional —el popular BASIC, por ejemplo— habría

que definir las variables, teniendo cuidado con su tipo y definir cómo podemos evaluar si alguien es padre o no de alguien y ver cómo comparamos las cadenas de caracteres y... otro sinfín de detalles que en PROLOG obviamos.)

Además, otra ventaja de PROLOG es que su sintaxis es muy cómoda y sencilla, y que sus expresiones son casi como en castellano. Más concretamente, podemos construir una «base de datos» en PROLOG(*) introduciendo las siguientes proposiciones:

```
CarlosI es-padre-de FelipeII
FelipeH es-padre-de CarlosI
```

y la siguiente regla de inferencia (es decir, regla de deducción):

```
x es-abuelo-de z si x es-padre-de y
                  & y es-padre-de z
```

y pasar a preguntar:

```
es (CarlosI es-abuelo-de FelipeII) ?
es (FelipeH es-abuelo-de FelipeII) ?
quién (x : x es-abuelo-de FelipeII) ?
```

(la última expresión se leería: «quién es x tal que x es el abuelo de Felipe II»; es decir, ¿quién es el abuelo de Felipe II?).

Las respuestas a estas tres preguntas son evidentes para una persona cualquiera, pero ¿cómo trabaja PROLOG? Para aclararlo, vamos a intro-

---

* El ejemplo que sigue está escrito en una versión castellana de micro-PROLOG que, aunque no sea muy utilizable en general, sirve para esta primera presentación: volveremos a hablar en detalle de las versiones disponibles de PROLOG y micro-PROLOG.

ducir alguna proposición más a nuestra base de datos (con Margarita de Parma, hija natural de Carlos I y su hijo Alejandro Farnesio, que sucedió a Juan de Austria como gobernador de los Países Bajos, con Isabel Clara Eugenia, hija predilecta de Felipe II, regente (pero no gobernadora) de los Países Bajos y con el propio Juan de Austria, también gobernador de los Países Bajos):

```
Carlos I es-padre-de FelipeII
Carlos I es-padre-de MargaritaP
Carlos I es-padre-de JuanA
Felipe II es-padre-de IsabelCE
MargaritaP es-madre-de AlejandroF
AlejandroF fue-gobernador
JuanA fue-gobernador
IsabelCE fue-regente
```

(obsérvese que han aparecido una nueva relación, el predicado «es-madre-de» y dos predicados adicionales: «fue-gobernador» y «fue-regente»).

Además, debemos completar la definición del abuelo con el nuevo predicado aparecido («es-madre-de»), quedando:

```
x es-abuelo-de z si x es-padre-de y
                 & y es-padre-de z
x es-abuelo-de z si x es-padre-de y
                 & y es-madre-de z
```

Si ahora queremos saber quién fue una persona cuyo abuelo fue Carlos I y fue gobernador de los Países Bajos, preguntaremos:

```
quién (x : CarlosI es-abuelo-de x & x fue-gobernador)
```

PROLOG procesará nuestra pregunta de izquierda a derecha e intentará averiguar quién cumple la primera condición indicada: CarlosI es-abuelo-de x. Para ello accede a las reglas de inferencia de que dispone y ve que

esa condición es equivalente a que se cumpla una cualquiera de las dos siguientes (nuevos subobjetivos a obtener):

CarlosI es-padre-de y & y es-padre-de z (1)
o bien CarlosI es-padre-de y & y es-madre-de z

Toma la primera de ellas y ve, en la base de datos, si hay algún elemento «y» que cumple esta primera condición; hay tres: Felipe II, Juan de Austria y Margarita de Parma; averigua si el primero de ellos (el primer valor «y» correspondiente a un hijo de Carlos I) cumple la segunda condición: en efecto «FelipeII es-padre-de IsabelCE». Por tanto, ya tenemos un elemento x que cumple CarlosI es-abuelo-de x. Pero, desgraciadamente, IsabelCE no cumple la segunda condición que imponemos en la pregunta, pues no fue gobernadora de los Países Bajos. Entonces PROLOG tiene que volver atrás *(backtracking* se llama el proceso) para seguir con alguno de los casos posibles que ha desechado anteriormente.

Toma entonces la segunda opción (JuanA) que cumplía «CarlosI es-padre-de y», pero este elemento no cumple «y es-padre-de z», pues en la base de hechos no aparece JuanA como padre de nadie.

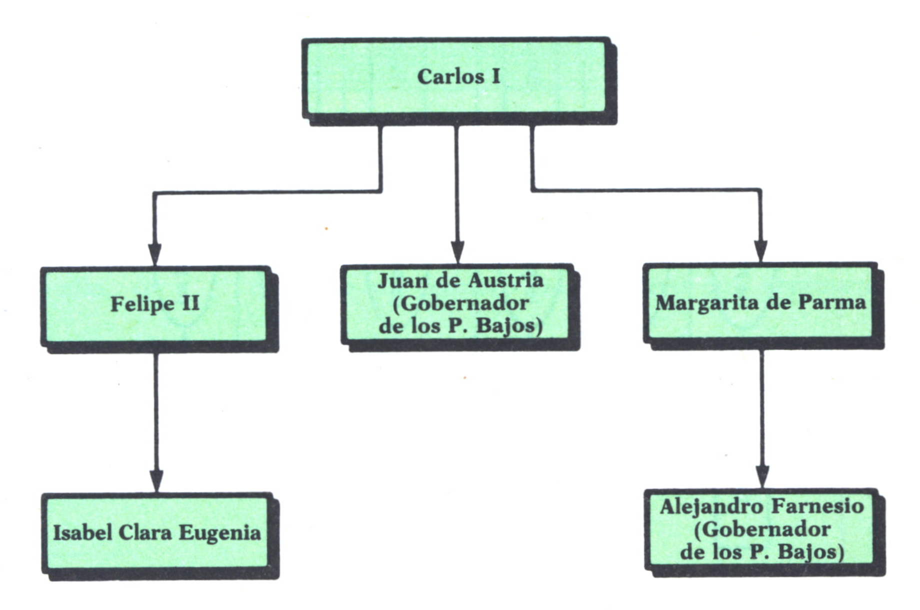



*Fig. 1. Descendientes de Carlos I.*

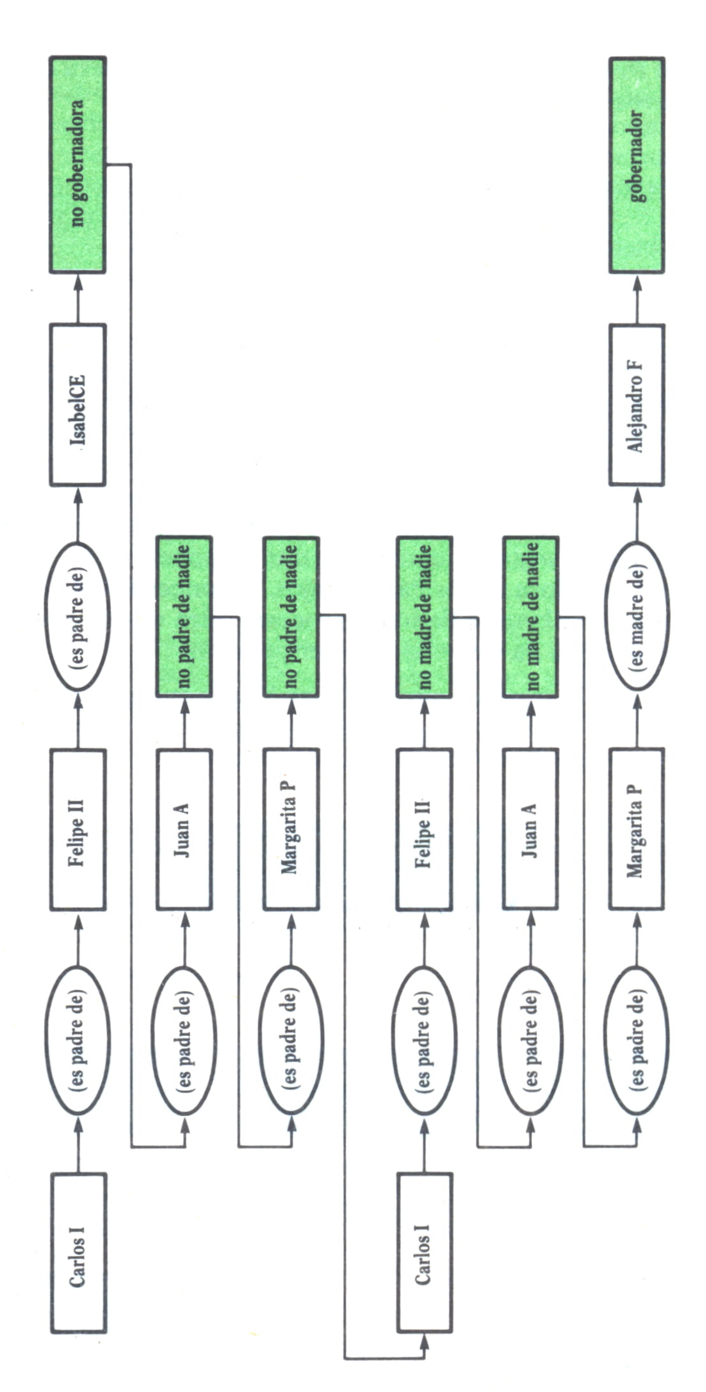



*Fig. 2. Proceso de Backtracking.*

Así, vuelve *(backtrack)* a Margarita de Parma que había cumplido la primera condición impuesta: CarlosI es-padre-de y. Para este nuevo valor de «y», preguntamos por la segunda parte de las condiciones impuestas como subobjetivos en (1) y vemos que MargaritaP no cumple tampoco, pues «MargaritaP es-padre-de z» no es cierto para ningún z. Esta rama del proceso se ha agotado sin éxito y hay que volver atrás *(backtracking)* preguntando por la segunda opción (subobjetivo) de las previstas en (1). Ahora PROLOG verá que «CarlosI es-padre-de MargaritaP» y «MargaritaP es-madre-de AlejandroF» (habiendo comprobado y desechado «CarlosI es-padre-de FelipeII» y «CarlosI es-padre-de JuanA», pues ninguno de los dos cumple «y es-madre-de z»).

Con este nuevo valor obtenido comprueba la segunda condición de la pregunta inicial y ve que, en efecto, Alejandro Farnesio fue gobernador de los Países Bajos; entonces PROLOG contesta:

```
x = AlejandroF
```

y como, después de las oportunas comprobaciones y vueltas atrás, descubre que no hay ningún otro elemento que responda a la pregunta, añade:

```
no hay (mas) respuestas.
```

Estos conceptos son los que el propio Colmerauer (*), coinventor del PROLOG, sintetiza en lo que llama el «reloj del PROLOG» (o máquina del PROLOG) y que aparece en la figura 3.

| | | |
|---|---|---|
| En ella, | «i» | contiene un número no negativo que representa el tiempo. |
| | «Ci» | contiene las restricciones (o condiciones) que deben ser satisfechas en el tiempo (momento) «i». |
| | «Ti» | incluye la secuencia de términos que no han sido eliminados aun en el momento «i». |
| | «Ri» | es el número de la regla elegida en el tiempo «i». |

---

* «Proceedings of the Eighth International Joint». *Conference on artificial Inteligence.* Karlsruhe. W. Germany. Tomo 1, pág. 496.

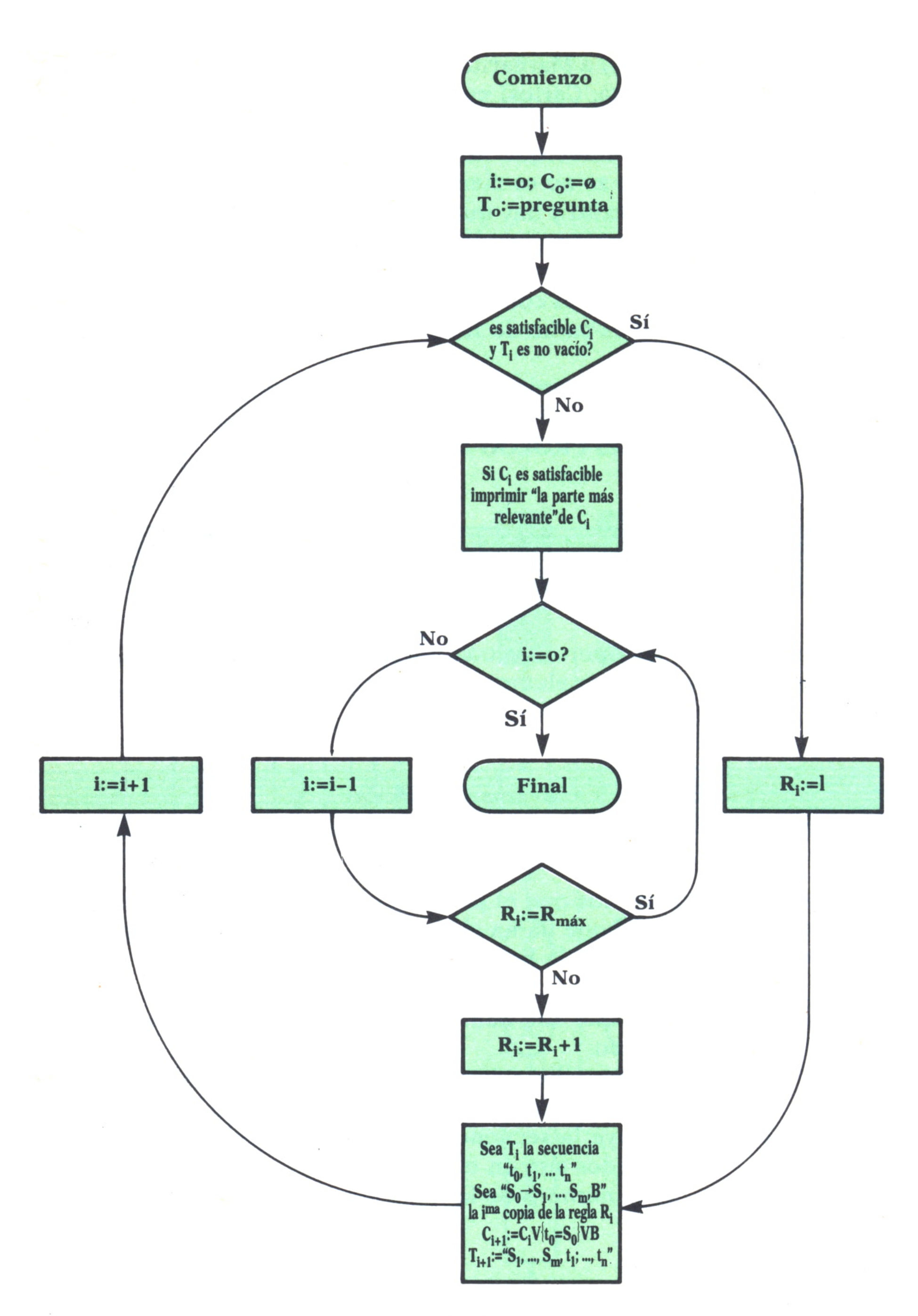



*Fig. 3. El reloj del PROLOG.*

«Rmax» indica el número de la última regla.
«la parte más relevante de Ci» significa la subcondición (en forma reducida) que se refiere a las variables que aparecen en la pregunta.

El funcionamiento de la máquina viene descrita por los dos círculos de la figura 3. En el exterior, el proceso gira una vuelta en el sentido de las agujas del reloj, mientras el tiempo («i») se incrementa en una unidad; en el interior se produce un giro en sentido contrario a las agujas del reloj, mientras el tiempo decrece una unidad. Se puede cambiar la progresión del tiempo pasando de uno a otro a través de uno de los dos «puentes» de un solo sentido que hay uniendo ambos círculos.

La ejecución de un programa PROLOG consiste en responder a una pregunta representada por un término (al principio se almacena en To). Todas las respuestas que se van computando son restricciones o condiciones en las cuales el término representa hechos específicos. Comenzamos el proceso por el par (Co, To); Co está vacío y To es la pregunta de comienzo. Cada vuelta que se da al círculo exterior incrementa la condición vigente actualmente «Ci» y transforma la secuencia «Ti». Si la regla que se utiliza ya contiene una condición B, esta condición se añade al juego de condiciones elementales de que disponemos. Concluye el proceso en cuanto se genera una condición no satisfecha o cuando la secuencia «Ti» queda vacía: es el momento en que se realiza el paso de *backtracking* (vuelta atrás) para tomar otras reglas. Si en algún momento, durante el proceso de giro del «reloj», «Ci» es satisfactorio, se imprime la respuesta (y vuelve atrás para localizar respuestas adicionales).

Para concluir esta breve introducción a PROLOG incluimos un programa escrito en el dialecto más popular de los existentes para microordenadores pequeños (el microPROLOG) y referente al «análisis» de una «base de hechos» geográfica (*).

*a)* Se utilizan las relaciones:

```
<ciudad> capital-de <provincia>
<provincia> provincia-de <autonomía>
<ciudad> situación (<posición-vertical> <posición-horizontal>)
```

referidas estas posiciones a su situación en un mapa imaginario. Como nombres de individuos se utilizarán las capitales de las provincias, las pro-

* Inspirado en el manual del micro-PROLOG preparado por K. L. Clark, F. G. McCabe y J. R. Ennais.

vincias de tres autonomías españolas y estas tres autonomías (Asturias, Rioja y País Vasco).

*b)* La base de datos podría ser:

Oviedo capital-de Asturias
Bilbao capital-de Vizcaya
SSebastian capital-de Guizpuzcoa
Vitoria capital-de Alava
Logroño capital-de Rioja
Vizcaya provincia-de PaisVasco
Guipuzcoa provincia-de PaisVasco
Alava provincia-de PaisVasco
Asturias provincia-de AsturiasAut
Rioja provincia-de RiojaAut
Oviedo situacion (15 5)
Bilbao situacion (16 12)
SSebastian situacion (17 15)
Vitoria situacion (13 13)
Logroño situacion (11 14)

*c)* Con todo esto, podríamos formular preguntas del tipo:

1. ¿Qué ciudades están al norte de Vitoria?
   Wich (x : x situacion (y z) and Vitoria situacion (Y Z) and Y LESS y)
   o bien en la versión castellana de microPROLOG:
   que (z : z situación (x y) e Vitoria situación (X Y) e X es menor que x)

(Nótese que en la versión castellana del microPROLOG de Spectrum la conectiva lógica «and» se ha traducido por «e» en vez de «y», para no confundirla con esta variable.)

2. ¿Hay alguna provincia del País Vasco cuya capital esté al Norte de Logroño y al Sur de Oviedo?
   is (x provincia-de PaisVasco and y capital-de x and y situacion (z X) and Logroño situacion (Y Z) and Oviedo situacion (x1 y1) and Y LESS z and z LESS x1)

o en castellano:

¿x provincia-de PaisVasco e y capital-de x e y situacion (z X) e Logroño situacion (Y Z) e Oviedo situacion (x1 y1) e Y es-menor-que z e z es-menor-que x1)

3. ¿En qué ciudad y autonomía hay una ciudad al sur y al este de Bilbao?
wich (x y : x provincia-de y and z capital-de x and z situacion (X Y) and Bilbao situacion (Z x1) and X LESS Z and x1 LESS Y)

y en castellano:

que (x y : x provincia-de y e z capital-de x e z situacion (X Y) e Bilbao situacion (Z x1) e X esmq Z e x1 esmq Y)

(utilizando la abreviatura «esmq» en vez de la frase completa es-menor-que).



## FUNDAMENTOS DE PROGRAMACION EN PROLOG

Vamos a examinar ahora con más detalle los elementos constitutivos del lenguaje PROLOG y los conceptos de base de la programación lógica en general y de la programación con PROLOG en particular. Para los ejemplos que vayan apareciendo utilizaremos la sintaxis que proporciona el «traductor» SIMPLE incluido en el intérprete de micro-PROLOG (igual que se ha hecho en la introducción de este capítulo). Esta sintaxis es más sencilla, lo que permite al lector no experto centrarse en los conceptos, sin dificultades de lectura de los programas. No se utilizará la versión castellana a partir de aquí porque no está disponible en todos los microordenadores usuales y entendemos que sería muy útil que el lector fuera introduciendo en su ordenador los ejemplos que le ofrecemos, para aclarar conceptos y para que se vaya familiarizando con el lenguaje, su sintaxis y el modo de programar con él en su microordenador; la versión original de SIMPLE, por otro lado, sólo requiere memorizar una decena de palabras inglesas. Hablaremos después de cómo crear fácilmente una versión castellana de SIMPLE, de la sintaxis estándar del micro-PROLOG, de las diferentes versiones existentes de PROLOG, etc.



## Elementos constitutivos del programa

Un programa PROLOG es un conjunto de aseveraciones o frases que contienen información. Es como una base de conocimientos organizada en forma de sentencias individuales.

Los elementos básicos constitutivos de estas frases o proposiciones son los «objetos», las «propiedades» y «relaciones», y las conectivas lógicas que realizan la ligazón de los anteriores elementos.

Las frases o proposiciones sencillas son simples aseveraciones en que aparece(n) un(os) sujeto(s) y algo que se dice de ese(esos) sujeto(s) (a esto que «se dice» del o de los sujetos se designa en lógica con el nombre de «predicado»). Estas frases constituyen los «hechos» del programa. Por ejemplo:

```
la-casa es-grande
la-casa está-a-la-derecha-de-el río
la-casa está-entre el-río la-colina
```

En el primer caso, aparece un sujeto («la-casa») y una propiedad que se le atribuye (el predicado monádico o unario «es-grande»); en la segunda frase aparecen dos sujetos («la-casa» y «el-río») y en medio una relación (el predicado diádico o binario «está-a-la-derecha-de»); en la tercera, tres sujetos («la-casa», «el-río» y «la colina») y otra relación (otro predicado —poliádico también— «está-entre»).

Desde el punto de vista formal, las propiedades (un predicado con un sujeto) y las relaciones (un predicado con varios sujetos) reciben el mismo tratamiento: en conjunto se les llama «relaciones». Se admite incluso una relación sin sujetos. En cuanto a los «sujetos» de estas frases, se suelen llamar en PROLOG «elementos» o «argumentos»; a veces, también, «objetos». Los elementos pueden ser simples y compuestos. Los elementos simples pueden ser, además, de dos tipos: constantes y variables. Las constantes están formadas por letras (o letras y el guión, si se unen palabras) formando «nombres» o por números, con o sin decimales. En los ejemplos anteriores, la-casa, el-río y la-colina son nombres compuestos (o sea, elementos simples y constantes). Los elementos compuestos, a su vez, están formados por otros elementos y se llaman listas. Una lista es un conjunto de elementos (elementos simples o listas) separados por blancos y encerrados entre paréntesis. Se admiten, por tanto, listas que contienen listas. Se admite, asimismo, la lista vacía «()». La siguiente lista tiene cuatro

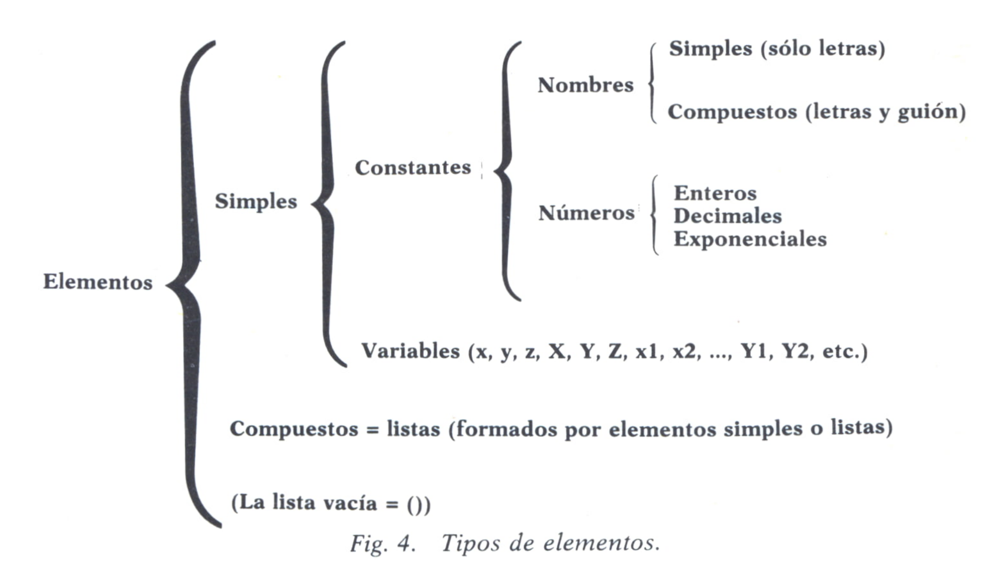



*Fig. 4. Tipos de elementos.*

elementos: una constante numérica, una variable, una lista formada por dos constantes y una variable y la lista vacía

```
(2 X2 (mínimo límiter Y) ())
```

Las frases compuestas se construyen ligando varias frases simples. Las conectivas lógicas utilizadas en PROLOG son el condicional «if» (es decir, «si» condicional), la conjunción «and» o «&» (en castellano es «y») y la negación «not» (que se corresponde con el castellano «no»). También está prevista la utilización de la disyunción «or», pero en una primera aproximación no es necesario su uso (por lo demás, de sintaxis un poco más complicada), pues en vez de poner dos frases simples unidas por «or» se consigue, generalmente, el mismo efecto incluyendo las dos frases: en su proceso de búsqueda de soluciones el analizador de PROLOG examinará la una y después la otra aceptando la afirmación correspondiente si se cumple cualquiera de ellas.

La estructura general, por tanto, de una frase será la que se muestra en la figura 5; en ella aparecen varias frases simples, un condicional y dos conjunciones. Aparte de estos elementos, también podría aparecer la negación, como en

```
X MAYOR-QUE 25 if not 25 LESS X and not 25 EQ X
```

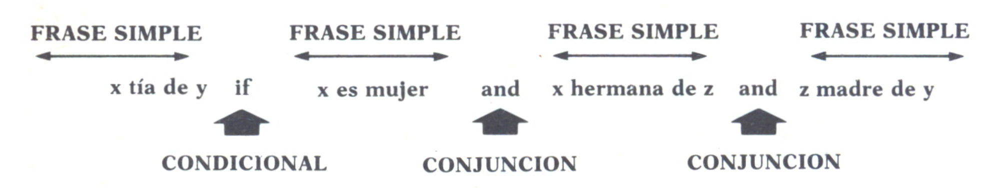
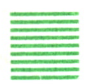



*Fig. 5. Estructura de una frase compuesta.*

donde LESS (con mayúsculas precisamente) es una palabra predefinida que significa «menor-que» y EQ representa la igualdad (veremos con detalle más adelante estas palabras predefinidas del PROLOG).



## Tipos de notación

Las frases del programa se pueden escribir en varias formas (manteniendo su significado). Se utilizan tres tipos diferentes de notación: prefija, infija y posfija.

En los anteriores ejemplos hemos utilizado la notación «infija»; es decir, la relación aparece entre los sujetos detrás del primer elemento u objeto y antes de los restantes, «la-casa está-entre el-río la-colina». En esta frase aparece la relación «está-entre» que vincula tres elementos: la-casa, el-río y la-colina. Esta notación es especialmente clara en los casos de relaciones con dos elementos: «la-casa está-a-la-derecha del-río».

Pero esto mismo se podría haber escrito en notación prefija (la relación o predicado se pone delante y los elementos detrás, como una lista; es decir, entre paréntesis y separados por espacios). Así podíamos haber escrito

```
está-entre (la-casa el-río la-colina)
está-a-la-derecha-de (la-casa el-río)
es-grande (la-casa)
```

Existe otro tercer modo de escribir una frase, que se utiliza básicamente para definir las propiedades de los elementos: es la notación posfija (primero el sujeto y después el predicado)

```
la-casa es-grande
```

Se puede utilizar cualquiera de estas notaciones, y en cada caso elegir la que resulte más clara, o se puede escribir (lo que es más usual) habitualmente en forma prefija. Esta notación (sin paréntesis) es la usada en micro-PROLOG, cuando no se utiliza SIMPLE.



## Construcción de una base de hechos

Los anteriores elementos descritos son suficientes para construir un sencillo programa en micro-PROLOG, pero debemos ver ahora cómo introducirle los datos al ordenador y cómo manejarlos (con ayuda del traductor adicional **SIMPLE**).

Para el manejo de la base de hechos (conjuntos de frases que constituyen el programa **PROLOG**) se dispone de las siguientes órdenes: «add» (añade), «accept» (acepta), «delete» (borra), «kill» (elimina), «edit» (edita) y «list» (lista).

Las dos primeras sirven para ir creando la base de conocimientos: en efecto, cuando cargamos **PROLOG** en el ordenador y nos da el mensaje de aviso (el «prompt» &.) cargando después el programa traductor **SIMPLE** (que vuelve a dar el prompt &.), podemos decir

```
add (Segovia está-en Castilla-León)
```

y esta frase («Segovia está-en Castilla-León) quedará incorporada a nuestra base de hechos. A continuación introduciremos otra frase

```
add (FelipeII fue-rey)
```

y otra

```
add (la-casa está-entre el-río la-colina)
```

Si, por el contrario, queremos introducir varias frases semejantes utilizaremos «accept» (acepta) para decirle que «nos acepte» varias frases acerca de una misma relación. Por ejemplo,

```
accept está-a-la-derecha-de
```

con lo cual a partir de ahora utilizará como nuevo prompt «está-a-la-derecha-de.», y podemos introducir varias frases, así:

accept está-a-la-derecha-de
*está-a-la-derecha-de.* (la-casa el-río)
*está-a-la-derecha-de.* (Luis Antonio)
*está-a-la-derecha-de.* (Iglesia Ayuntamiento)
*está-a-la-derecha-de.* end

donde las palabras en cursiva las escribe el propio intérprete, a modo de prompt, para pedir datos de otra nueva frase, recordando de paso la relación que estamos tratando en este momento. Con la palabra «end» (final) concluimos el proceso de la instrucción accept.

Si, una vez que hayamos cargado nuestra base de hechos, queremos saber qué hay en ella (para su corrección o simple curiosidad) no tenemos más que indicarle al traductor

```
list all
```

es decir, «lista todo» y nos dará un listado de toda la base de hechos

```
Segovia está-en Castilla-León
FelipeII fue-rey
la-casa está-entre el-río la-colina
está-a-la-derecha-de (la-casa el-río)
está-a-la-derecha-de (Luis Antonio)
está-a-la-derecha-de (Iglesia Ayuntamiento)
```

Si, por el contrario, queremos listar únicamente las proposiciones que contengan la relación «está-a-la-derecha-de», escribiremos

```
list está-a-la-derecha-de
```

y obtendremos la siguiente relación:

```
está-a-la-derecha-de (la-casa el-río)
está-a-la-derecha-de (Luis Antonio)
está-a-la-derecha-de (Iglesia Ayuntamiento)
```

El listado que se obtiene cuando se le pide una lista completa *(list all)* aparece con todas las afirmaciones referentes a una relación agrupadas y en el orden en que han sido introducidas.

Este tipo de listado es de mucha ayuda para la utilización de otra forma que puede adoptar una frase con «add»: si queremos añadir una frase dentro del grupo de las que tienen la relación «está-a-la-derecha-de», pero en la tercera posición (pasando, por tanto, la frase que teníamos en tercera posición a la cuarta) podemos escribir

```
add 3 (el-cuchillo está-a-la-derecha-de el-plato)
```

de tal modo que si ahora hacemos

```
list está-a-la-derecha-de
```

obtenemos

```
está-a-la-derecha-de (la-casa el-río)
está-a-la-derecha-de (Luis Antonio)
está-a-la-derecha-de (el-cuchillo el-plato)
está-a-la-derecha-de (Iglesia Ayuntamiento)
```

Si deseamos eliminar frases de la base de hechos, podemos utilizar uno de los dos comandos siguientes: delete o kill.

El primero de ellos, «delete» (borra) suprime una sola frase de la base de conocimientos, y el otro, «kill» (elimina), suprime todas las proposiciones que haya en la base de datos con la relación que se le indica.

Por ejemplo, si decimos

```
delete (está-a-la-derecha-de (el-cuchillo el-plato))
```

o bien

```
delete está-a-la-derecha-de 3
```

eliminará (en ambos casos igual) la tercera frase que hay en el grupo de las que tienen la relación «está-a-la-derecha-de».

Si, por el contrario, decimos

```
kill está-a-la-derecha-de
```

se eliminarán las cuatro frases que hay en ese grupo, de modo que un «list all» después dará como resultado, solamente:

```
Segovia está-en Castilla-León
FelipeII fue-rey
la-casa está-entre el-río la-colina
```

Por último, si lo que queremos es modificar una frase, disponemos de una palabra adicional que es «edit» (editar); se usa con el nombre de la relación en cuyo grupo está la frase que queremos cambiar y el número de orden dentro de ese grupo. Así, si antes de hacer el «kill» que acabamos de indicar (o una vez restaurada la base de datos) tecleamos

```
edit está-a-la-derecha-de 3
```

aparecerá en la pantalla la frase

```
3 (está-a-la-derecha-de (el-cuchillo el-plato))
```

y podemos modificarla con las facilidades de edición de que dispongamos en nuestro ordenador.



## Almacenamiento y recuperación de datos

Pero la base de hechos anterior (ya escrita y depurada) queremos guardarla para su posterior utilización: debemos escribir la primitiva «save» (guarda) con el nombre del fichero donde queremos que se almacene nuestra base de conocimientos. De este modo,

```
save nuevo-fich
```

guardará todo el contenido actual de la base de conocimientos en el fichero «nuevo-fich».

En sentido contrario, para volver a cargar el fichero en el área de trabajo, escribiríamos

```
load nuevo-fich
```

aunque antes deberíamos haber borrado lo que hay en la base de datos, para no duplicar la información. (En efecto, la orden «load» no borra el contenido de la base de conocimientos antes de cargar los nuevos datos.)

En el desarrollo de estas manipulaciones es útil saber que SIMPLE mantiene un «diccionario» de todas las relaciones presentes. Este diccionario es accesible mediante el argumento «dict». Si escribimos

```
list dict
```

aparecerá un listado de las relaciones que hemos utilizado:

```
está-en dict
fue-rey dict
está-entre dict
está-a-la-derecha-de dict
```

de modo que cada relación tiene a la derecha la palabra «dict»: podemos pensar que dict es como una propiedad (en notación posfija). De hecho, se puede manejar como una relación más y tiene, por tanto, sentido escribir

```
delete dict 2
```

o bien

```
delete (fue-rey dict)
```



## Utilización sencilla de los conocimientos

Hemos visto hasta ahora cómo construir y almacenar una base de hechos. Veamos ahora cómo utilizarla. Para hacer preguntas a la base de conocimientos (es decir, extraer información de ella) disponemos de cuatro primitivas adicionales: «is» (es), «all» (todos), «wich» (cuál o qué) y «one» (uno sólo).

La utilización de estas primitivas es bastante semejante al uso que se hace de ellas en el lenguaje natural. La primitiva «is» puede adoptar dos formas: con constantes o con variables. Una pregunta del primer tipo sería:

```
is (Segovia está-en Castilla-León)
```

(que recibirá la respuestas YES, puesto que esa afirmación está en nuestra base de hechos). Una pregunta con variables puede ser

```
is (Segovia está-en X)
```

que tendrá también la respuesta YES, pues hay una relación que se adecúa a ese esquema, una vez sustituido X por el valor «Castilla-León».

También se puede preguntar, no si existe una afirmación dada o no, sino para qué elemento se cumple la relación: es decir, preguntar «qué elemento X cumple que X está-en Castilla-León» o bien «qué X hay tal que X está-en Castilla-León» y en micro-PROLOG:

```
wich (X : X está-en Castilla-León)
```

a lo que el intérprete responderá:

```
Segovia
No (more) answers
```

dando el elemento que cumple la proposición propuesta (es decir Segovia) y añadiendo «No (more) answers» (No —más— respuestas disponibles). Si hubiera varias respuestas, se darían todas las que cumplen la condición.

Otro modo de hacer la pregunta es utilizar la primitiva «all» (todos) que es equivalente a «which»:

```
all (X : X está-en Castilla-León)
```

o bien sustituir la pregunta por otra que comience por la primitiva «one» (uno sólo), que produce el mismo efecto que «all» pero sólo da una respuesta (la primera que encuentra) y pregunta si se quiere seguir: si se le contesta afirmativamente continuará la búsqueda; en caso contrario concluye.

Naturalmente, puesto que el(los) argumento(s) de la pregunta es(son) una proposición de las que tenemos o podemos tener en la base de he-

chos, con o sin variables, se pueden hacer preguntas más sofisticadas mediante las conectivas lógicas (como se describió al presentar las frases simples de la base de conocimiento). Así se puede preguntar:

```
wich (X : X está-entre el-río la-colina &
          X está-a-la-derecha-de el-río)
```

o bien,

```
all (X Y : X está-entre Y la-colina &
           está-a-la-derecha-de (la-casa Y))
```

Además, se dispone de dos palabras primitivas muy útiles en las preguntas: «defined» (definida) y «reserved» (reservada), que se refieren a esas dos propiedades aplicadas a diferentes elementos. Así, si con la pequeña base de datos que estamos manejando preguntamos:

```
is (está-en defined)
```

el sistema contestaría YES (sí), pues esa relación está, efectivamente, definida en el sistema. Si preguntamos:

```
is (está-en reserved)
```

la respuesta sería NO, pues «está-en» no es una de las palabras reservadas (primitivas) de micro-PROLOG. Por el contrario, respondería YES si le preguntásemos:

```
is (reserved reserved)
```

El argumento «dict», ya conocido, puede ser utilizado en las preguntas para saber si tenemos una relación en el diccionario o no. La pregunta:

```
is (está-en dict)
```

obtendría una respuesta YES. Por el contrario,

```
is (más-grande-que dict)
```

generaría un NO por parte del sistema.



## Inclusión de reglas

Toda la labor realizada hasta ahora es básica, pero poco fructífera: el sistema ha almacenado la información que le hemos facilitado y ha contestado a algunas preguntas que le hemos hecho (con búsquedas exhaustivas a veces, y hechas las preguntas de un modo muy cómodo, eso sí). Pero debemos añadir más información para que el sistema de inferencias de PROLOG funcione y obtenga nuevos datos a partir de los que le hemos facilitado: estas informaciones adicionales adoptan la forma de reglas. Una regla de inferencia es una frase PROLOG en que se incluyen dos frases simples o compuestas (con sus relaciones correspondientes) vinculadas por «if». Por ejemplo, podemos establecer:

```
X está-entre el-río la-colina if X está-a-la-derecha-de el-río
```

Con esto estamos marcando o definiendo una propiedad general que permitirá al sistema obtener conclusiones que no se le hayan dado, pero que sean deducibles (lógicamente) de los datos que se le han facilitado. Por ejemplo, si hubiéramos introducido la regla anterior, de la afirmación «la-casa está-a-la-derecha-de el-río» se podría deducir «la-casa está-entre el-río la-colina», y no hubiera sido necesario teclear dicha aseveración (o si no disponíamos de ella, la hubiéramos podido obtener).

El conjunto de hechos y reglas de inferencia constituye un programa en PROLOG y se puede decir que la parte más «creativa» de él son precisamente las reglas, por lo que es usual llamar a los lenguajes del tipo de PROLOG lenguajes «basados en reglas».

Las reglas se introducen en nuestra base de conocimientos al igual que los simples hechos, mediante la primitiva «add», se eliminan con «delete», etcétera.



# REFERENCIA DE PRIMITIVAS

Hasta aquí hemos estado viendo la estructura general de un programa en PROLOG y sus elementos constitutivos. Para realizar algunas funciones (borrar, añadir, preguntar...) hemos utilizado algunas primitivas de micro-PROLOG o SIMPLE (palabras reservadas que realizan funciones concretas ajenas a la propia estructura del programa).

Además de éstas, existen en PROLOG otras palabras reservadas que se utilizan para operaciones concretas (manejo de listas, realización de operaciones aritméticas, control del proceso de búsqueda, etc.).

Vamos a presentar a continuación las más generales, organizadas en función de la actividad a la que se destinan.



## Operaciones con números

Para realizar operaciones aritméticas están disponible (al menos) cinco primitivas en PROLOG: «SUM» (suma); «TIMES» (por, multiplicado por, producto); «LESS» (menor que); «INT» (entero) y «EQ» (igual).

Aunque parezcan pocas funciones para el manejo de números, la flexibilidad de PROLOG hacen que sean suficientes y que el resto de herramientas que el lenguaje proporciona ayude en la programación de los cálculos; a pesar de que, desde luego, PROLOG no ha sido diseñado para el proceso de números, sino de otro tipo de conocimientos e informaciones (que suelen venir dados en forma literal, básicamente).

Obsérvese, por otro lado, que las primitivas indicadas aparecen con letras mayúsculas; la razón es que estas funciones son primitivas de micro-PROLOG que SIMPLE pasa directamente al intérprete sin traducir (como sucedía con «add», «delete», etc.).

La palabra SUM se define diciendo que es cierto:

```
SUM (X Y Z)
```

si Z = X + Y. Es decir, la respuesta a:

```
is (SUM (2 3 5)).
```

será YES.

Se puede utilizar también, con el mismo sentido, en preguntas como:

```
which (x: SUM(2 3 x))
```

que dará como resultado:

```
5
```

A partir de esta primitiva SUM se pueden definir (por parte del usuario) otro conjunto de operaciones aritméticas que pueda necesitar.

Por ejemplo, se puede introducir la regla:

```
MASSUMA(X1 X2 X3 Z) IF SUM (X1 X2 Y1) & SUM(Y1 X3 Y2)
                       & SUM(Y2 X4 Z)
```

Que define la suma de tres números. O bien:

```
which(x: SUM(x 1.25 3.75))
```

que da como resultado la *diferencia* entre 3.75 y 1.25, es decir:

```
2.5
```

También se puede definir de un modo general:

```
diferencia (x1 x2 z) IF SUM (x2 z x1)
```

Los números utilizados en PROLOG pueden ser enteros (positivos *—que no deben llevar signo—*, o negativos con signo), decimales o en coma flotante (una secuencia de dígitos decimales con su parte entera y su parte decimal seguida de la letra E (de exponencial) y un entero).

Para las multiplicaciones se utiliza la primitiva TIMES, que se puede definir diciendo que será cierto:

```
TIMES(x v z)
```

si sucede que z = x*y. Se pueden hacer preguntas del tipo:

```
all(z: TIMES(2 3 z))
```

que dará como resultado:

```
6
```

o bien, para realizar divisiones:

```
all(z: TIMES(2 z 6))
```

que dará como resultado, naturalmente:

```
3
```

Se puede, sin embargo, definir expresamente la división:

```
DIV(x y z) if TIMES(y z x)
```

Para la comparación de los números se utilizan dos primitivas: LESS representa el concepto «menor que» y EQ se utiliza para representar en PROLOG la igualdad. Así, la pregunta:

```
is(LESS(2 7))
```

tendrá como respuesta:

```
YES
```

y la pregunta:

```
is (EQ(6 5))
```

obtendrá la respuesta:

```
NO
```

Es sumamente interesante también la primitiva INT, que representa a los números enteros. Exactamente su función es comprobar si un número es entero o, caso de que sea decimal, obtener su parte entera. El cálculo de la parte entera de un decimal la realiza INT siempre por defecto; es decir, la parte entera de 3.25 es 3 y como parte entera de –3.25 da –4. Esta primitiva adopta dos posibles formas: como predicado monádico (con un solo sujeto, un número) sirve para comprobar si un número es entero. Adopta en este caso la notación posfija. Por ejemplo, en la pregunta:

```
is(6 INT)
```

la respuesta será:

```
YES
```

Por el contrario, en la forma de predicado binario se utiliza para generar la parte entera de un número decimal:

```
is(-3.25 INT x & x EQ -4)
```

dará como resultado:

```
YES
```

y la expresión:

```
which(x: 3.25 INT x)
```

producirá la siguiente respuesta:

```
3
```



## Manejo de listas

Tal como dijimos en su momento, una lista es una sucesión de elementos separados por blancos y encerrada entre paréntesis. La lista puede estar vacía, (), o puede contener, a su vez, otras listas como elementos de ella:

```
(x 2.25 a (2 x) (5 X Y))
```

Cada lista hay que considerarla como un elemento en el interior de otra lista o como argumento de alguna función. Dentro de cada lista es útil

distinguir en numerosas ocasiones el primer elemento de la lista (llamado «cabeza» de la lista) del resto de los elementos que en ella aparecen (resto o «cola» de la lista). Con el fin de precisar estos detalles se utiliza la expresión (x|y) para representar a una lista cuya cabeza es «x» y cuya cola es «y». Por tanto, si estamos considerando la lista (a b c d), la asignación de valores que realizará PROLOG para (x|y) es: x es el elemento a; y es la lista (b c d).

La barra de separación, |, se suele leer «seguido de», de tal modo que se acepta la expresión (x y | z) para representar una lista formada por los elementos x e y «seguido de» otro elemento z (que puede ser a su vez una lista). Como además el elemento z puede ser la lista vacía, la expresión anterior representa a todas las listas formadas al menos por dos elementos.

En ocasiones, en vez de la barra, |, se escribe una a modo de barra partida formada por dos puntos alargados, ¦.

Es importante distinguir la lista como una unidad, de los elementos que la componen. Por ejemplo, en las expresiones:

```
Luis es-de Familia-Lopez
Juan es-de Familia-Lopez
Carmen es-de Familia-Lopez
(Ramon Juan) es-de Familia-Lopez
(Maria (Carmen Luis)) es-de Familia-Lopez
```

vemos que la relación «es-de» admite un sujeto que sea unitario y una lista (incluso en el último caso una «lista de listas»). Si preguntamos:

```
is (Juan es-de Familia-Lopez)
```

la respuesta sería YES, pero si la consulta es:

```
is (Ramon es-de Familia-Lopez)
```

la contestación será NO, mientras que obtendremos YES al preguntar:

```
is((Ramon Juan) es-de Familia-Lopez)
```

Si preguntamos:

```
wich (X: X es-de Familia-Lopez)
```

nos contestará el intérprete una relación del estilo siguiente:

```
Luis
Juan
Carmen
(Ramon Juan)
(Maria (Carmen Luis))
```

Podemos, incluso, hacer consultas del tipo:

```
wich (X : (X1 X2) es-de X)
```

y el sistema contestará con las familias (en nuestro caso sólo tenemos a la familia-López) a las que pertenecen elementos formados por dos elementos. O bien introducir:

```
wich (X : X EQ (X1 X2) & X es-de Familia-Lopez)
```

O sea, elementos de la familia-López que se adecúen al esquema (X1 X2). (Obsérvese que la primitiva EQ sirve para comparar listas igual que números.) La respuesta sería:

```
(Ramon Juan)
(Maria (Carmen Luis))
```

Si la pregunta es:

```
wich (X : (X1 (X X2)) es-de Familia-Lopez)
```

la respuesta sería:

```
Carmen
```

(estamos preguntando por listas que tengan, concretamente, la estructura dada (X1 (X X2)); es decir, dos elementos el segundo de los cuales es, a su vez, una lista de dos elementos).

O podemos querer obtener, por ejemplo, el primer elemento (cabeza) de la lista que está en segundo lugar en las listas de elementos que pertenecen a la familia-López, sin saber la longitud de esta segunda lista; la consulta sería, en este caso:

```
wich (X : (X1 (X|X2)) es-de Familia-Lopez)
```

y la respuesta:

```
Carmen
```

(Obsérvese que, en este caso, hemos representado la lista que está en segundo lugar como (X|X2) indicando que nos interesa la cabeza «X», pero que después, a modo de cola, hay otro elemento genérico «X2» que puede ser toda una lista, un elemento, la lista vacía...).

Pero en ocasiones nos interesa la operación contraria del análisis de listas o la extracción de elementos de una lista, es decir, la unión o concatenación de dos listas. Para ello se utiliza la relación «append». Esta palabra no es una primitiva del lenguaje, pero suele estar predefinida como relación (en caso contrario, el usuario puede introducirla de acuerdo con la definición que damos más adelante, al hablar de la recursión). Esta relación tiene tres parámetros: los dos primeros son las listas a concatenar y el tercero la lista resultante; así:

```
append (X Y Z)
```

significa que Z es la concatenación de la listas X e Y. Concretamente:

```
append ((Ramon Juan) (Teofilo) (Ramon Juan Teofilo))
```

será verdad.

Se puede utilizar en preguntas:

```
is (append ((Ramon Juan) (Teofilo) (Ramon Juan Teofilo)))
```

obtendrá la respuesta YES. Mientras que:

```
all (X : append ((Ramon Juan) (Teofilo) X)
```

tendrá como respuesta:

```
(Ramon Juan Teofilo)
```

Se puede añadir un elemento «César» en la cabeza de la lista anterior:

```
all (X : append ((Cesar) (Ramon Juan Teofilo) X)
```

dando:

```
(Cesar Ramon Juan Teofilo)
```

(Obsérvese que «append» maneja listas, por lo que hemos tenido que hacer César lista, para incluirlo en append como (César)).

Incluso, se puede usar «append» para descomponer listas:

all (X : append ((Cesar) X (Cesar Ramon Juan Teofilo)))

dará como resultado la «cola» de la lista que aparece en tercer lugar en «append»:

(Ramon Juan Teofilo).



# RECURSION Y OTRAS TECNICAS

## Iteración y recursión

En los ejemplos anteriores hemos visto cómo examinar la «cabeza» de una lista y detectar o escribir que esa cabeza está formada por un solo elemento o por dos, tres..., etc. Sin embargo, nos queda detrás una «cola» de lista que, por ahora, no sabemos manejar bien. Por otro lado, tampoco podemos saber cuántos elementos tiene una lista.

Para realizar esta tarea es necesario utilizar un proceso iterativo de análisis: es decir, un proceso paso a paso que vaya «separando» cada elemento de la lista.

Por ejemplo, si queremos averiguar todos los individuos que pertenecen (relación «es-de») a la «familia-López» independientemente de que estén como tales individuos aislados o dentro de una lista habría que hacer una pregunta del tipo:

all (X :X es-de Familia-Lopez)

que nos dá todos los elementos individuales de la relación «es-de», pero también:

all (X : (X X1) es-de Familia-Lopez)
all (X : (X2 X) es-de Familia-Lopez)

y así para todo el tipo de listas que pueden aparecer en la base de hechos. Para evitar esto (normalmente, incluso, ni conocemos cuáles son las diversas estructuras de lista en juego), deberíamos hacer un examen iterativo de una lista genérica donde observemos cada elemento para ver si cumple la condición correspondiente. Un problema semejante surge cuando se quieren «contar» los individuos de una lista; teóricamente es fácil: se toma el individuo que sea la «cabeza» de la lista y se anota uno; se toma después la cola de la lista como nueva lista, y se vuelve a separar la cabeza añadiendo uno a la cuenta que llevamos; se vuelve a tomar como lista base la cola resultante y se vuelve a separar la cabeza anotando una unidad más, etcétera. Sin embargo, este proceso genérico debe ser programado como una actividad iterativa (programar un paso genérico que se repite para todos los elementos de la lista). En PROLOG no existe predefinida ninguna estructura de control que realice esta tarea, como es usual en la mayoría de los lenguajes (bucles DO, FOR, etc.) y es necesario programarlo.

Se suele utilizar una técnica de recursión para resolver estas actividades iterativas. La recursión consiste básicamente en que en la ejecución de un procedimiento se llama, bajo unas condiciones concretas, al mismo procedimiento.

Hay que tener cuidado cuando se diseña un proceso recursivo para que el programa no entre en un «bucle» infinito (la función se está llamando a sí misma constantemente), sino que esté adecuadamente controlado el final del proceso, y el bucle concluya. Por ejemplo, en el caso en que estemos examinando los diferentes elementos de una lista (para controlar alguna condición dada o simplemente para «contarlos») tomando elemento a elemento, se puede cerrar el proceso recursivo preguntando por la condición de que la lista restante sea vacía.

Así, podríamos definir una función «longitud de una lista» (en inglés «length») mediante la regla:

```
X length Y if X EQ (X1|X2) & X2 length Z & SUM (Z 1 Y)
```

donde decimos que la longitud (length) de X es Y si la longitud de X2 (cola de la lista) es Z y resulta que Y = Z + 1; es decir, que la longitud de la cola resultante cuando se le separa el primer elemento de cabeza. Aún se puede condensar más esta definición escribiendo:

```
(X1|X2) length Y if X2 length Z & SUM (Z 1 Y)
```

donde se ha separado, en la lista X cuya longitud queremos averiguar, la cabeza (X1) de la cola (X2). Esta definición es un caso típico de recursión, pues para definir la «length» de X es necesario antes «calcular» la «length» de su cola (X2) y sumarle una unidad; pero para el cálculo de la longitud de X2 habrá que aplicar, de nuevo, la misma definición añadiendo una unidad a la longitud de la cola de X2 (pongamos que sea X3), y así sucesivamente. Sólo existe una pequeña dificultad en la regla que estamos manejando y es que no está previsto un control de terminación del proceso recursivo: cuando la lista cuya longitud haya que calcular (después de un número determinado de pasos de iteración) sea la lista vacía, al ir a calcular la «cola» de la lista, se producirá un error. Para evitar este problema (¡problema eterno de los procesos recursivos!) hay que definir, aparte, la longitud de la lista vacía (que parece normal definirla como cero). Escribiremos, por tanto:

```
() length 0
(X1|X2) length Y if X2 length Z & SUM (Z 1 Y)
```

Ahora, en cada fase del «cálculo de longitud» de una lista o sublista empezará el intérprete «procesando» la primera línea: si la lista no es vacía, pasará a la segunda línea; si la lista es, precisamente, la lista vacía, le asignará longitud 0 y avanzará al siguiente paso.

De este modo, si introducimos la expresión:

```
all (Y : (M1 (M2 M3) M4) length Y)
```

el intérprete verá que la lista dada (M1 (M2 M3) M4) no es vacía y realizará la «identificación» de esta lista con el «esquema» dado en la segunda expresión haciendo X1 igual a M1 y X2 igual a ((M2 M3) M4). (Este proceso se llama «unificación»). Intentará entonces calcular Z como longitud de X2 y encuentra que no «sabe» la longitud de ((M2 M3) M4), por lo que deja sin evaluar este valor (provisionalmente queda pendiente la asignación de valor para este primer Z obtenido; lo llamaremos Z1). A continuación, comprueba que la nueva lista (la «cola» X2; es decir ((M2 M3) M4) no es vacía y aplica la «definición» de longitud haciendo X1 igual a (M2 M3), cabeza de la lista, y X2 igual a (M4), cola de la lista. Tampoco sabe la longitud de (M4), por lo que dejará sin evaluar una nueva longitud (Z2). Aplica de nuevo la «definición» tomando X1 igual a M4 y X2 igual a la cola

resultante que es la lista vacía. La longitud de esta cola (Z3) no la conoce y vuelve a aplicar la definición. Pero ahora la lista que maneja es ya la lista vacía, por lo que asigna como longitud de esta lista el valor 0. Así, pues, ahora sabe que Z3 = 0 y, por tanto, según las expresiones de SUM que ha ido dejando pendientes, asignará valores Z2 = Z3 + 1 = 1; Z1 = Z2 + 1 = 2 y, por fin, Y = Z1 + 1 = 3, devolviendo como valor de Y el número 3.

Este proceso de llamadas recurrentes a sí misma de la función «length» es un procedimiento usual utilizado en la programación en PROLOG, y es enormemente potente: sólo hay que tener cuidado con establecer adecuadamente la condición de terminación del proceso.

Es usual que la primitiva «length» aparezca definida en el intérprete de PROLOG; pero si no es así, puede usted introducirla mediante la definición dada anteriormente.

Del mismo modo que la primitiva «length» se puede definir la primitiva «append» (de la que ya hemos hablado), si en su PROLOG no está incluida. El resultado de unir (append) una lista X con otra Y para dar la lista Z es igual que si tomamos el último elemento de X y lo unimos con Y y al resultado de esa unión le anteponemos los restantes elementos de X: por ejemplo:

```
all (Z : append ((M1 M2 M3) (M4 M5) Z)
```

dará como resultado:

```
(M1 M2 M3 M4 M5)
```

que es la consecuencia de unir X (es decir, (M1 M2 M3)) con Y (o sea, (M4 M5)). Pero el proceso lo podemos pensar como si tomamos el último elemento de X (M3) y lo unimos con Y para dar (M3 M4 M5) y al resultado le ponemos delante los elementos suprimidos en X, es decir (M1 M2). Y también para hacer esta última unión, tomar el último elemento (M2) y unirlo con el resto (M2 M3 M4 M5) anteponiendo después el elemento suprimido (M1). Todo este proceso lo podemos expresar mediante una definición (recursiva) de la primitiva «append»:

```
append ((X1|X2) Y (X1|Z2)) if append (X2 Y Z2)
```

En esta regla, se aparta la cabeza de X (es decir, X1) y se une el resto (X2) con Y para dar Z2. Pero para hacer este append (el de X2 con Y para dar Z2) habrá que volver a «separar» la cabeza de X2 y unir con Y el resto y así sucesivamente. Esto representa el proceso que hemos descrito antes, pero formulado al revés. Sólo queda establecer la condición de terminación del proceso. Esta conclusión se debe producir cuando en la lista que está en la posición X no quedan más elementos para unir con Y, es decir, cuando la lista X sea vacía. Obviamente la unión (append) de la lista vacía con una lista cualquiera Y dará como resultado la propia lista Y. El conjunto completo de la definición de append será, por tanto:

```
append (() Y Y)
append ((X1|X2) Y ( X1|Z2)) if append ((X2 Y Z2))
```

También se puede definir de modo recursivo la primitiva «belongs-to» («pertenece a») que establece la propiedad de los miembros de una lista de «pertenecer a» ella. (Normalmente esta primitiva estará definida en su PROLOG; si no, puede usted introducirla. Naturalmente, si va usted a introducir estas primitivas, puede definirlas con los nombres en castellano: «es-de-longitud» o simplemente «longitud» en vez de «length», «une» en lugar de «append», «pertenece-a» por «belongs-to», etc. Nosotros usamos los nombres ingleses por ser los más universalmente utilizados.)

Para poder definir el concepto de «pertenece-a» para los elementos de una lista de tres elementos pondríamos sencillamente: «X pertenece-a Y si X es igual al primer elemento de Y (su cabeza) o igual a su segundo elemento (cabeza de su cola) o igual a su tercer elemento (cola de la cola)». Pero si no sabemos la longitud de la lista Y, tendremos que usar una expresión genérica y acudir a la recursión para, mediante un proceso iterativo, ir examinando la coincidencia del elemento X con los distintos elementos de Y. Concretamente, esto se puede hacer definiendo: «X pertenece a Y si X es la cabeza de Y o bien si X pertenece a la cola de Y». La formulación de esta frase en PROLOG se realizará mediante dos relaciones alternativas:

```
X belongs-to (X Y2)
X belongs-to (Y1 Y2) if X belongs-to Y2
```

Así, pues, si preguntamos:

```
all (X : X belongs-to (A B C D))
```

PROLOG responderá:

```
A
```

de acuerdo con la primera regla; y si pedimos más respuestas (recuérdese que «all» va dando las respuestas una a una)

```
B
C
D
no (more) answers (no (más) respuestas)
```

También se pueden definir recursivamente otras primitivas no utilizables en el manejo de listas, sino en operaciones aritméticas: por ejemplo, la primitiva «factorial».

Como el lector sabe, el factorial de un número entero «n» (se suele escribir n!) es el resultado de multiplicar «n» por «n–1», por «n–2», etc.; es decir, el producto de todos los enteros hasta el «n» inclusive. Además, el factorial de 1 es 1. Y se suele definir que el factorial de 0 es, también, 1. No podemos definir de un modo explícito el factorial de «n» multiplicando los «n» primeros enteros menores o iguales a «n» (pues «n» puede ser cualquier número). Es necesario hacerlo de un modo iterativo formulando que el factorial de «n» es igual a «n» multiplicado por el factorial de «n–1». Es decir, si X1 es una unidad inferior a X, el factorial de X es igual a X multiplicado por el factorial de X1. O bien en PROLOG (diciendo que «X factorial Y» significa que Y es el factorial de X: Y = X!):

```
x factorial de Y if
        TIMES (X Y1 Y) &
        X1 factorial Y1 &
        SUM (X1 1 X)
```

Esta definición es válida para cualquier X menor que 1, pero producirá error para X = 1 (pues PROLOG no podría calcular el factorial de X1 en este caso). Hay, por tanto, que definir este valor (cerrando, de este modo, el ciclo de recurrencias que se produce al ir calculando los factoriales de

números cada vez una unidad menores, X1). La definición completa será, por tanto:

```
0 factorial 1
1 factorial 1
X factorial Y if
                1 LESS X &
                TIMES (X Y1 Y) &
                X1 factorial Y1 &
                SUM (X1 1 X)
```

## Búsqueda bajo condiciones más complejas

Aparte de las preguntas que podíamos realizar mediante las primitivas «is», «wich», «all»... que producían respuestas simples (un elemento cada vez), se puede conseguir que la contestación a una pregunta sea una lista. Esto sucede si se realiza la pregunta con la primitiva «isall» («es-todo»). La respuesta es *una lista* con todos los elementos que cumplen la condición: todas las respuestas que hubiéramos obtenido si hubiéramos hecho la pregunta con «all» puestas en forma de lista. Así, por ejemplo, la pregunta:

```
wich (X : X isall (X1 : X1 está-entre X2 X3 & X1 está-a-la-derecha-de X2))
```

obtendría la respuesta:

```
(la-casa)
```

pues el único elemento que cumple la condición de X1 descrita anteriormente es «la-casa» y lo devuelve en forma de lista. Si la respuesta a la condición de X1 hubiera sido (con otra base de hechos):

```
la-casa
el-pajar
el-granero
no (more) answers
```

la correspondiente respuesta de «isall» sería:

```
(la-casa el-pajar el-granero)
```

Naturalmente, «isall» se puede usar no sólo en preguntas, sino formando parte de reglas del programa. Por ejemplo:

```
(X Y) padres-de Z if
                    Z isall (Z1 : X padre-de Z1 &
                                  Y madre-de Z1)
```

Otra condición muy interesante es la que se construye con la primitiva «forall». El formato general de la condición «forall» es:

```
(forall condición1 then condición2)
```

donde condición1 y condición2 son dos condiciones simples o compuestas (con la conjunción and, & —o sea, «y»—). Hay que leer la frase anterior como «para todos los elementos que cumplen condición1 se cumple condición2». Por ejemplo, se puede definir el subconjunto de un conjunto dado del siguiente modo:

```
X sublista-de Y if
      (forall X1 pertenece-a X then X1 pertenece-a Y)
```

y a partir de esta definición establecer la siguiente:

```
X mismos-elementos-que Y if
             X sublista-de Y &
             Y sublista-de X
```

Por otro lado, se pueden hacer preguntas más complejas que las establecidas hasta ahora introduciendo junto a la condición ya definida (and, &), la disyunción (o) y la negación (no).

La disyunción (A o B) se formula como:

```
either A or B
```

(es decir, «bien A o B»). Como ya hemos comentado, es equivalente normalmente a poner ambas condiciones sucesivamente:

```
A
B
```

puesto que PROLOG evalúa «la una o la otra» indistintamente (y sucesivamente). Así, es equivalente escribir:

```
X EQ Carmen if (X X1) es-de Familia-López
X EQ Carmen if (X2 (X X3)) es-de Y &
                    Y EQ Familia-López
```

que escribir:

```
X EQ Carmen if (either (X X1) es-de Familia-López or
                    (X2 (X X3)) es-de Y &
                    Y EQ Familia-López)
```

También se pueden establecer preguntas complejas introduciendo la negación mediante la partícula «not». La condición «not» se cumplirá si *no se cumple* la correspondiente condición afirmativa formulada según las reglas descritas. Así, se puede escribir:

```
wich (X : CarlosI es-abuelo-de X &
            not X fue-gobernador)
```

que pregunta por los nietos de Carlos I que no fueron gobernadores de los Países Bajos.



## Control de la búsqueda de soluciones

Ya hemos comentado cómo PROLOG realiza la búsqueda de soluciones, para cualquier pregunta que se le propone, siguiendo una rama del árbol y volviendo hacia atrás *(backtracking)* hasta el último nudo examinado, para seguir después por otra rama, volver atrás, etc., hasta encontrar todas las soluciones. Esta técnica tan fructífera en general, puede ser sumamente tediosa en ocasiones. Si conocemos la estructura de la base de hechos que manejamos o las condiciones en que se va a producir la búsqueda, podemos controlar este proceso de «vuelta atrás» mediante el símbolo «/». En efecto, esa barra inclinada impide la continuación del proceso de búsqueda hacia atrás. Se utiliza como una condición simple más dentro de cualquier condición compleja y unida, por tanto, al resto de la frase mediante una conjunción («&»): cuando PROLOG, en su evaluación de soluciones encuentra dicho símbolo detiene el proceso de búsqueda hacia atrás y da las soluciones obtenidas. La barra no tiene por qué estar al final de la condición: si está en medio de las condiciones, sólo se efectuará la búsqueda hacia atrás para las condiciones detrás de la barra.



# DIALECTOS Y VERSIONES

Existen numerosas versiones de PROLOG e, incluso, variantes del lenguaje (dialectos y derivados), pero vamos a presentarlos reunidos en tres grupos:

a) Por un lado, están las diferentes versiones «oficiales» de PROLOG. Este lenguaje ha sido soportado básicamente, en sus comienzos, y apoyado por el grupo del profesor Colmerauer en la Universidad de Marsella-Luminy. Han ido produciéndose en ese grupo, o en otros que trabajan relacionados con él, numerosas y sucesivas mejoras al lenguaje. Actualmente está disponible ya la versión 2 del PROLOG II para ordenadores VAX, Mackintosh e IBMPC. Esta versión ha sido desarrollada en Pascal, lo que le da flexibilidad (el propio usuario puede modificar y ampliar las características del lenguaje) y portabilidad. Según la profesora García Serrano y su grupo (Facultad de Informática de Madrid), las características básicas del lenguaje son:

Permite la continuidad entre la forma de entender y de especificar los problemas para su programación, no obliga al determinismo y dispone de

variables generalizadas. Además, la unificación de términos es automática, así como la estrategia de búsqueda primera en profundidad con *back-trackting* al anterior punto de elección. La computación es dirigida por el objetivo y el control se realiza con metarreglas y predicados definidos.

Entre los predicados definidos del PROLOG II v2 se destaca el predicado «freeze», que permite retardar la ejecución de objetivos y el predicado predefinido «dif», que presenta la particularidad de que no es evaluado hasta que sus argumentos han sido unificados.

Con respecto a las estructuras de datos aparece en esta implementación además de las listas, la estructura de árbol (infinito o no) con un código interno económico que permite parametrizar predicados, agrupar objetivos o definir listas de longitud determinada.

Entre las utilidades del sistema se encuentran predicados que comunican con el sistema operativo, un editor de línea, predicados de modificación y estructuración de reglas en los llamados «mundos», predicados de inserción y eliminación de reglas (todos ellos utilizables desde el sistema PROLOG o desde el programa). Permite, además, la comunicación con otros lenguajes (Pascal, Basic, Fortran).

Esta versión de PROLOG es exigente tanto en tiempo como en memoria, pero se espera que con técnicas de paralelismo, compilación u otras, se consiga una mayor eficiencia en su ejecución.

Se ha utilizado para realización de prototipos de Sistemas Expertos de tipo medio (< = 500 reglas) y su operación ha sido interesante para esta misión, así como para modelos de comprensión del lenguaje natural con el método de las gramáticas de metamorfosis definidas por Colmerauer.

Incluimos a continuación tres programas pequeños en PROLOG para que puedan observarse las escasas diferencias existentes con los programas vistos hasta ahora.

La definición de «append» en PROLOG será (compárese con la versión ya presentada de la misma relación para micro-PROLOG):

```
append ([], L, L).
append ([X | L1], L2, [X | L3])
    :— append (L1, L2, L3)
```

en donde aparecen dos expresiones: la primera define el resultado de unir la lista vacía a otra lista cualquiera. La segunda reduce la concatenación de dos listas [X | L1] y L2 a la concatenación correspondiente de la primera lista sin su cabeza, con la segunda lista.

El cálculo del discriminante de una ecuación de segundo grado se puede hacer mediante la expresión:

```
discriminante(A, B, C, D)
   :— mult(B, B, Balcuad), mult(A, C, S1),
      mult(4, S1, S2), add(S2, D, Balcuad)
```

donde Balcuad es el cuadrado de B; S1 es un subtotal resultado de multiplicar A*C; S2 equivale a 4*A*C y D se obtiene como aquel número que hay que sumar a S2 para obtener B, es decir, B – 4*A*C.

Por último, podemos examinar el programa que el propio Colmerauer presentó en el congreso de IJCAI-83 para resolver el conocido criptograma SEND + MORE = MONEY, donde cada letra puede representar un dígito cualquiera (de 0 a 9) y la asignación de valores tiene que ser tal que la suma sea correcta. El programa consta de tres partes: la primera enseña, sencillamente, a sumar (los números se representan con un punto por medio: por ejemplo, 15=1.5; 9=0.9). La segunda define la secuencia de valores sin repetición (eliminando las letras duplicadas). Con estas definiciones se puede pasar a la tercera parte, donde se hace realmente el cálculo de la solución. Si se pregunta después por «resultado (X,Y,Z)», la respuesta será X=9.5.6.7; Y=1.0.8.5; Z=1.0.6.5.2, como el lector habrá calculado ya.

Primera parte

```
suma (0.0,x,x)→
menor-que-veinte (x);
suma (x',y,z')→
más-uno (x,x')
suma (x,y,z)
más-uno (z,z');
menor-que-veinte (0.0) ;
menor-que-veinte (y)→
más-uno (x,y);
más-uno (0.0,0.1) ;
más-uno (0.1,0.2) ;
más-uno (0.2,0.3) ;
más-uno (0.3,0.4) ;
más-uno (0.4,0.5) ;
más-uno (0.5,0.6) ;
más-uno (0.6,0.7) ;
más-uno (0.7,0.8) ;
más-uno (0.8,0.9) ;
```

```
más-uno (0.9,1.0) ;
más-uno (1.0,1.1) ;
más-uno (1.1,1.2) ;
más-uno (1.2,1.3) ;
más-uno (1.3,1.4) ;
más-uno (1.4,1.5) ;
más-uno (1.5,1.6) ;
más-uno (1.6,1.7) ;
más-uno (1.7,1.8) ;
más-uno (1.8,1.9) ;
```

Segunda parte

```
sin-repetición (vacía)→;
sin-repetición (u.l)→
quita (u,l)
sin-repetición (l);
quita (u,vacía)→;
quita (u,v.l)
quita (u,l),
{u=v};
```

Tercera parte

```
resultado (S.E.N.D,M.O.R.E,M.O.N.E.Y)
sin-repetición (S.E.N.D.M.O.R.Y.vacía)
posible (r1,0,0,M,0)
posible (r2,S,M,0,r1)
posible (r3,E,0,N,r2)
posible (r4,N,R,E,r3)
posible (0,D,E,Y,r4)
{S<>0,M<>0};
posible (0,u1,u2,u3,r)
suma (0.u1,0.u2,r.u3);
posible (1,u1,u2,u3,r)
suma (0.u1,0.u2,x)
más-uno (x,r.u3);
```

(Se han utilizado $r_1$, $r_2$, $r_3$ y $r_4$ para representar los «acarreos» —arrastre de una unidad o de ninguna para la suma parcial siguiente cuando la suma de dos dígitos individuales rebasa la decena—.)

b) Por otro lado, existen numerosas versiones del lenguaje PROLOG que se ofrecen independientes o integradas en otros conjuntos de programación o entornos más complejos.

## QUINTUS PROLOG

Versión desarrollada por Quintus Computer Systems, Inc., aporta básicamente dos ventajas: una, más fácil manejo (para los angloparlantes) pues sus instrucciones se manejan con formatos parecidos al propio idioma inglés, lo que lo aproxima a toda persona no versada en la nomenclatura de la lógica formal; y sus mecanismos de control incorporados que facilitan la creación cómoda de prototipos. Xerox ofrece una versión de QUINTUS PROLOG, mejorada en prestaciones gráficas, con sus estaciones de trabajo de I.A., junto al INTERLISP y otras facilidades de desarrollo.

## QUTE

Desarrollado en la Facultad de Ciencias de la Universidad de Tokio, es una amalgama de LISP y PROLOG. Es válida en QUTE cualquier expresión PROLOG y cualquier expresión LISP. Cada una de ellas es tratada por la parte correspondiente de QUTE, además, la parte LISP y la parte PROLOG se llaman una a otra recursivamente.

Se ha implementado en un ordenador VAX bajo sistema operativo UNIX.

Aunque en QUTE una expresión PROLOG puede contener, a modo de subexpresiones, una expresión LISP y viceversa, las expresiones simbólicas pueden jugar el doble papel de datos y programas, como sucede en LISP. Todas estas capacidades producen un lenguaje más claro y una mayor libertad de expresión y representación del conocimiento.

## ARITY/PROLOG

Es una versión desarrollada por Arity Corp., básicamente para ordenadores que trabajen bajo MS/DOS. Incluye acceso aleatorio a los archivos de datos y manejo de pantallas. Como dispone de una facilidad para manejar memoria virtual, puede llegar a procesar aplicaciones de gran tamaño. Por ello, es muy útil cuando se trabaja en ordenadores no grandes (IBM PC/AT, por ejemplo) pero con una gran RAM disk y disco duro de gran tamaño. Dispone de un compilador bastante eficaz y algunas otras características sofisticadas de PROLOG como son la notación infija con precedencia de operador y las gramáticas con cláusulas definidas (DCG.) En la úl-

tima versión disponible (versión 4.0) permite la definición de varios «mundos» (particiones independientes de la base de datos) y, por tanto, admite definir en «mundos» separados el «código» y los «datos», lo cual es especialmente útil si se manejan grandes espacios de memoria virtual, pues aumenta enormemente la eficiencia de proceso del sistema.

## TURBO PROLOG

Ha sido desarrollado por Borland International Inc., autora de otros lenguajes «turbo» muy populares. Aporta una facilidad enorme de manejo mediante una presentación en pantalla a través de cinco «ventanas» y un conjunto cómodo y flexible de menús y comandos predefinidos. La sintaxis de TURBO PROLOG es bastante diferente de la estándar de PROLOG y, por ello, más cómoda de manejar y leer. El programa en TURBO PROLOG se divide en diferentes secciones de declaración de los diversos elementos (alguna de estas secciones son opcionales): sección de «dominios» donde se declaran las listas y tipos definidos localmente por el usuario; la sección de «dominios globales» permite que los tipos de datos en ella definidos sean utilizables por otros módulos de programa; la sección de «predicados» define el tipo de los predicados locales y el tipo y número de los argumentos que cada predicado debe utilizar; la sección de («predicados globales» cumple la misma misión para los predicados accesibles desde otros módulos de programa; puede incorporarse una sección de «objetivo» (goal) para incluir un objetivo a alcanzar (esta sección es opcional, y puede omitirse para ir dando los objetivos en tiempo de ejecución, pero debe incluirse si se compila el programa); la sección de «base de datos» especifica los predicados a los que se van a añadir «hechos» en tiempo de ejecución; la sección de «cláusulas» contiene los hechos y reglas que forman el cuerpo del programa. Contiene numerosos predicados predefinidos y un conjunto muy cómodo de comandos para manejo de archivos. Su compilador es bastante eficiente, y los programas compilados son rápidos. Se le acusa de no ser un PROLOG clásico, sino ser casi un híbrido de Pascal y PROLOG (se le llama, un tanto despectivamente, «TURBO PASLOG»). Sin embargo, es muy cómodo de manejar y bastante flexible. A modo de ejemplo de la programación en este lenguaje que se está haciendo tan popular, damos la versión TURBO PROLOG de la definición del factorial, y la versión completa del programa de las «torres de Hanoi».

*Definición del factorial de un número entero*

```
domains
   n, f = integer
```

```
predicates
  factorial (n,f)
clauses
  factorial (1,1).
  factorial (N,Res) if
    N > and
    N1 = N-1 and
    factorial (N1,FacN1) and
    Res = N*FacN1.
```

*Las Torres de Hanoi*

```
DOMAINS
  TIME, ROW, COL, NUMBE = INTEGER
PREDICATES
  hanoi( NUMBER )
  move( NUMBER, NUMBER, ROW, ROW, ROW, COL, COL, COL )
  inform( NUMBER, NUMBER, ROW, ROW, COL, COL )
  makepole( NUMBER, NUMBER, COL)
  delay() dd(TIME)
  move_vert(COL,NUMBER,ROW,ROW)
  move_horizon(ROW,NUMBER,COL,COL)
CLAUSES
  delay :- dd(100).
  dd(0):-!.
  dd(N):-N1=N-1,dd(N1).
  hanoi (N) :-
    N<=13,!,
    VB=2+6*N,VH=3+N,CV=N, CM=3*N, CH=5*N,
    STCOL=(79-6*N)/2, STROW=25-VH)/2,
    makewindow(1,7,7,"Hanoi",STROW,STCOL,VH,VB),
    makepole(N,N,CV),
    move(N,N,0,0,0,CV,CM,CH),
    cursor (0,0), write("Pulse una tecla"),readehar(_).
  hanoi(_):- write("máximo 13 disc's\n").
  move(H,1,HA,_,HC,CA,_,CH):-!,INFORM(H,1,HA,HC,CA,CH).
  move(H,N,HA,HB,HC,CA,CB,CC):-
      N1=N-1.
      HA1=HA+1,
      move(H,N1,HA1,HC,HB,CA,CC,CB),
      inform(H,N,HA,HC,CA,CC),
      HC1=HC+1,
      move(H,N1,HB,HA,HC1,CB,CA,CC).
  inform( H, N, H1, H2, C1, C2 ) :-
    C11=C1-N, C22=C2-N, NN-2*N,
    H11=H-H1, H22=H-H2,
    move_vert(C11,NN,H11,1),
    move_horizon(1,NN,C11,C22),
```

```
    move_vert(C22,NN,1,H22).
  makepole(_,0,_):-!.
  makepole(H,N,C):-HH=H-N,inform(H,N,HH,HH,C,C), N1=N-1, makepo-
  le(H,N1,C).
  move_vert(_,_,H,H):-!.
  move_vert(COL,SIZE,H1,H2):-H1<H2,!, /* mueve hacia arriba */
    H11=H1+1,
    field_attr(H11,COL,SIZE,112),
    field_attr(H1,COL,SIZE,7),delay,delay,
    move_vert(COL,SIZE,H11,H2).
  move_vert(COL,SIZE,H1,H2):-H1>H2,!, /* mueve hacia abajo */
    H11=H1-1,
    field_attr(H11,COL,SIZE,112),
    field_attr(H1,COL,SIZE,7),delay,delay,
    move_vert(COL,SIZE,H11,H2).
  move_horizon(_,_,H,H):-!.
  move_horizon(ROW,SIZE,C1,C2):-C1<C2,!, /* mueve a la derecha */
    C11=C1+1, HH=C1+SIZE,
    field_attr(ROW,HH,1,112),
    field_attr(ROW,C1,1,7),delay,
    move_horizon(ROW,SIZE,C11,C2).
  move_horizon(ROW,SIZE,C1,C2):-C1>C2,!, /* mueve a la izquierda */
    C11=C1-1, HH=C11+SIZE,
    field_attr(ROW,C11,1,112),
    field_attr(ROW,HH,1,7),delay,
    move_horizon(ROW,SIZE,C11,C2).
goal
  hanoi(6).
```

c) Sacamos aparte del grupo anterior una versión muy popular de PROLOG desarrollada específicamente para ordenadores personales: el micro-PROLOG, que es la versión más extendida en este tipo de equipos (y que es la utilizada en todo este capítulo por entender que es la más accesible al lector medio de este libro). La mayoría de las versiones existentes de micro-PROLOG incluyen un traductor adicional llamado SIMPLE: las palabras «add», «list», «if», etc., utilizadas hasta ahora, son comandos que SIMPLE traduce a la sintaxis estándar de micro-PROLOG. La sintaxis estándar de micro-PROLOG es más sencilla y compacta que la de SIMPLE, pero por ello mismo, más oscura y difícil de leer. Los conceptos básicos de proceso y las estructuras de programa son muy semejantes, por lo que ha sido útil utilizar SIMPLE en los ejemplos. Los comandos en micro-PROLOG se escriben siempre con mayúsculas (pero las variables pueden estar escritas con minúsculas). Como sabemos, en SIMPLE las palabras clave se escribían en minúsculas, excepto algunas (LESS, SUM, etc.) que tecleamos en mayúsculas precisamente porque son primitivas de micro-PROLOG que el traductor SIMPLE pasa al intérprete, sin más cambio.

Las reglas básicas de la sintaxis de micro-PROLOG (que difieren de la sintaxis de SIMPLE) son:

— Los elementos constitutivos del lenguaje son los átomos y cláusulas: un átomo es una aseveración básica de las que hemos llamado relación o propiedad, con sus argumentos correspondientes; en micro-PROLOG desaparecen las notaciones infija, prefija o posfija y se escribe simplemente la proposición como una lista con el predicado (la relación) en cabeza y los argumentos detrás, todo entre paréntesis. Así, en vez de

```
AlejandroF fue-gobernador
Juan es-de Familia-Lopez
la-casa está-entre el-rio la-colina
```

en micro-PROLOG hay que escribir:

```
(fué-gobernador AlejandroF)
(es-de Juan Familia-Lopez)
(está-entre la-casa el-rio la-colina)
```

lo cual resulta más difícil de leer (se parece menos al castellano usual), pero simplifica la escritura, al unificar la notación de todos los tipos de relación.

— El segundo elemento (más complejo) constitutivo del lenguaje es la cláusula. Una cláusula es el equivalente de la regla de derivación utilizada en SIMPLE. Formalmente, es una lista formada por átomos (o por un solo átomo). Una aseveración simple (predicado o relación más sus argumentos) es una cláusula formada por un solo átomo, aunque para ser considerada como tal cláusula por el intérprete debe ser convertida en lista (encerrándola entre paréntesis). Así,

```
(es-de Juan Familia-Lopez)
```

es un átomo, mientras que

```
((es-de Juan Familia-Lopez))
```

es una cláusula unitaria («singulete», se llama a veces). Si la expresión que consideramos no es una aseveración simple, sino una regla de derivación, del tipo

```
X está-entre el-rio la-colina if X está-a-la-derecha-de el-rio
```

la conversión en cláusula de micro-PROLOG consiste en suprimir el «if» y cambiar a átomos las proposiciones simples que forman la frase. Así, la cláusula correspondiente será:

```
((está-entre X el-rio la-colina) (está-a-la-derecha-de X el-rio))
```

donde aparece una lista formada por átomos entendiéndose que el primero es la frase que se coloca delante de «if» y el segundo la proposición que se escribe detrás. Si la frase es más complicada, por ejemplo del tipo:

```
MASSUMA(X1 X2 X3 Z) IF SUM(X1 X2 Y1) & SUM(Y1 X3 Y2)
                                      & SUM(Y2 X4 Z)
```

la cláusula correspondiente en micro-PROLOG será:

```
((MASSUMA  X1  X2 X3 Z) (SUM X1 X2 Y1) (SUM Y1 X3 Y2) (SUM Y2  X4
Z))
```

que, como se ve, sigue siendo una lista en la que el primer átomo es la proposición que precedía al «IF» y el resto de átomos son las frases que estaban tras el «IF» suprimiendo, también, la conjunción «AND» (o «&»).

— Algunas condiciones y primitivas más sofisticadas que se usan en SIMPLE tienen una «traducción» más compleja en micro-PROLOG:

**either...or...* se sustituye, simplemente, por el predicado OR con dos cláusulas (siempre entre paréntesis y separadas por un blanco para formar —a su vez— otra cláusula). Así, una frase del tipo:

```
(EQ X Carmen) OR(es-de (X X1) Familia-Lopez) ((es-de
     (X2(X X3)) Y) (EQ Y Familia-Lopez)))
```

se escribirá:

```
X EQ Carmen if(either (X X1) es-de Familia-Lopez
        or (X2(X X3))es-de Y & Y EQ Familia-Lopez)
```

donde aparece una cláusula formada por dos cláusulas (esta es la estructura del if); la segunda de estas cláusulas está formada por el OR y dos cláusulas: una simple y otra segunda formada por una lista de dos cláusulas (que corresponde con la conjunción —and— de estas dos cláusulas).

**not*. Aparece escrita en mayúsculas, con una interrogación detrás y teniendo como argumento a una cláusula; el conjunto se encierra entre paréntesis para constituir, a su vez, otra cláusula.

Así, la frase

```
not (la-casa está-entre el-rio la-colina)
```

aparece como

```
(NOT ? (está-entre la-casa el-rio la-colina))
```

* *isall.* Esta relación aparece como cabeza de una lista cuya cola está formada por la lista de respuestas, la lista de variables que han de cumplir las condiciones y la cláusula donde se expresan las condiciones (esta cláusula será una lista de cláusulas, si en la versión SIMPLE hubiéramos puesto una condición compuesta formada por varias, relacionadas entre sí por la primitiva «and»). Por ejemplo:

```
(X isall (X1 X2 X3 : X1 está-entre X2 X3 & X1 está-a-la-derecha-de
X3))
```

habrá que escribirlo:

```
(ISALL X (X1 X2 X3) (está-entre X1 X2 X3) (está-a-la-derecha-de
X1 X3)))
```

* *forall.* Aparece (como sucedía con OR) con dos argumentos correspondientes a los dos conjuntos de condiciones.

De este modo, la expresión de SIMPLE

```
(forall X1 está-entre X2 la-colina then X1 está-a-la-derecha-de X2)
```

se escribirá en micro-PROLOG:

```
(FORALL  (está-entre X1 X2 la-colina)) ((está-a-la-derecha-de  X1
X2)))
```

— Otras primitivas mantienen, prácticamente, su sintaxis: así, se usan igual que en SIMPLE las primitivas LIST, EQ, KILL, y, en general, la mayoría de las que hemos utilizado (en SIMPLE) con mayúsculas.

— Para añadir y borrar cláusulas de la base de hechos, se usan ADDCL (en vez de «add») y DELCL (en vez de «delete»), pero con la misma sintaxis. Sin embargo, si lo que queremos es simplemente añadir una cláusula (sin colocarla en un sitio predefinido ni ninguna otra de las opciones más sofisticadas que el «add» permite), se teclea sencillamente y micro-PROLOG la incorpora a su base de datos.

— Las funciones de escritura disponibles son P y PP. La primera escribe una frase (que aparece como cola de un átomo cuya cabeza es precisamente la función «P») y la segunda realiza la misma operación, pero haciendo una «vuelta del carro» a continuación.

Por ejemplo,

```
(P El valor del diámetro es X2)
```

escribirá la frase «El valor del diámetro es 5.2» si el valor de «X2» era 5.2, y quedará lista para seguir escribiendo en el mismo renglón, mientras que

```
(PP El valor del diámetro es X2)
```

escribirá la misma frase pero producirá, después, un salto al comienzo de la línea siguiente.

Para introducir datos se usa la primitiva «R» que espera datos desde el teclado y los va asignando a las variables que se le han puesto como argumentos hasta asignar valor a todas ellas. Por ejemplo, se puede preparar una expresión del tipo «(R X)» para aceptar un dato y asignárselo a la variable «X». Se podría construir un diálogo del siguiente modo:

```
& ? (P Cual es tu nombre) (R X) (P Hola X)
Cual es tu nombre
Luis
Hola Luis
```

donde la primera y la tercera líneas han sido tecleadas por el usuario, y la segunda y la cuarta han sido escritas por el intérprete de micro-PROLOG.

— La interrogación se hace mediante la primitiva «?» (que equivale a «is» de SIMPLE) y la respuesta será & (el propio «prompt» del sistema) si es YES o bien «?» (como respuesta de micro-PROLOG) si es NO.

— Los comentarios se introducen con «/*». O sea, una cláusula que comience por «/*» no será evaluada por el sistema. Así, se pueden documentar los programas, y este capítulo podríamos cerrarlo (si fuera un programa en micro-PROLOG) con el comentario:

/* Fin del capítulo tercero

# BIBLIOGRAFIA


## A. Libros básicos (y muy accesibles) sobre Inteligencia Artificial y sus lenguajes de programación

- *Inteligencia Artificial.* J. L. Aubert y R. Schomberg. Ed. Paraninfo, 1986.
- *LISP. El lenguaje de la Inteligencia Artificial.* A. A. Berk. Ed. Anaya Multimedia, 1986.
- *PROLOG. Programación y aplicaciones en Inteligencia Artificial.* A. A. Berk. Ed. Anaya Multimedia, 1985.
- *Micro-PROLOG. La lógica como lenguaje para la alfabetización informática.* R. Ennals. Ed. Díaz de Santos, S.A., 1987.
- *Introducción a LISP.* H. Farreny. Ed. Masson, S.A., 1986.
- *Inteligencia Artificial: conceptos y programas.* T. Hartnell. Ed. Anaya Multimedia, 1986.
- *LISP. Introducción al cálculo simbólico.* D. S. Touretzky. Ed. Díaz de Santos, S.A., 1986.

## B. Libros específicos sobre algún lenguaje

- *Programming Expert Systems in OPS5.* L. Brownston, R. Farrel, E. Kant, N. Martin. Ed. Addison-Wesley, 1985.
- *LISP. Cromemco instruction manual.* Ed. Cromemco Inc., 1980.
- *Smalltalk. Programming Handbook.* Ed. Digitalk Inc., 1986.
- *Performance & evaluation of Lisp Systems.* R. P. Gabriel. Ed. The MIT Press, 1986.
- *PROLOG. Manuel de Référence et d'utilisation.* Ph. Roussel. Faculté de Sciences de Luminy, Marsella, 1973.
- *LISP.* P. H. Winston y Horn. Ed. Addison-Wesley, 1981.

### C. Libros generales de aplicación de la Inteligencia Artificial y/o los lenguajes de programación


• *Research and Development in Expert Systems.* Ed. por M. A. Bramer, Ed. Cambridge University Press, 1985.

• *Artificial Intelligence. An Introductory Course.* A. Bundy. Ed. Edimburgh University Press, 1980.

• *Artificial Intelligence Programming.* E. Charniak, C. K. Riesnek, D. V. MacDermott. Ed. Lawrence Erlbaum Associates, 1980.

• *Applying Expert Systems in Business.* D. N. Chorafas. Ed. McGraw-Hill Book Company, 1987.

• *The handbook of Artificial Intelligence.* E. A. Feingenbaum y otros. Tres tomos. Ed. William Kaufman, 1982. (Especialmente el volumen II, para conocer el LISP y sus aplicaciones.)

• *Artificial Intelligence.* E. Rich. Ed. McGraw-Hill Book Company, 1983. (Varios lenguajes.)

• *LOGICS for Artificial Intelligence.* R. Turner. Ed. Ellis Horwood Ltd., 1984.

• *Artificial Intelligence.* P. H. Winston. Ed. Addison-Wesley Publishing Co., 1984.


---

*NOTA.* Aunque una de las aplicaciones más populares de la Inteligencia Artificial es la construccion de sistemas expertos, no se han incluido aquí las referencias de los libros que tratan este tema de un modo específico, pues hay un tomo de esta colección dedicado a este área. Lo mismo se podría decir de otras aplicaciones concretas de la Inteligencia Artificial: Visión Artificial, Criptografía, Juegos Inteligentes, etc.

**Libro en que se describen los lenguajes específicos para la «elaboración del saber» y los entornos de programación correspondientes.**

**El conocimiento de estos lenguajes, además de interesante en sí mismo, es sumamente útil para entender todo lo que la Inteligencia Artificial supondrá para el futuro de la Informática y está significando hoy en día.**